AFRL-RI-RS-TR-2016-068

RESOURCE ANALYSIS OF COGNITIVE PROCESS FLOW USED TO ACHIEVE AUTONOMY

LOCKHEED MARTIN

MARCH 2016

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2016-068 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
THOMAS E. RENZ
Work Unit Manager

/ S /
RICHARD MICHALAK
Acting Technical Advisor
Computing & Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* MARCH 2016	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* SEP 2014 – SEP 2015

4. TITLE AND SUBTITLE RESOURCE ANALYSIS OF COGNITIVE PROCESS FLOW USED TO ACHIEVE AUTONOMY	**5a. CONTRACT NUMBER** FA8750-14-C-0278
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) David Rosenbluth	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER R1HE
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lockheed Martin, 3 Executive Campus Drive, Cherry Hill, NJ 08002	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITB, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2016-068

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2016-0930
Date Cleared: 2 MAR 2016

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The current challenge of autonomy is to achieve a reasonable scaling between task breadth and system resources. As processing resources are a limiting factor for autonomous operations in complex environments, the incorporation of new enabling low power processing technologies into autonomous systems is important to overcoming current limitations and keeping pace with peer adversaries. With the increasing variety of processing technologies, the number of design choices for implementing end-to-end cognitive processing flows multiplies and the impact of these design decisions on efficiency and effectiveness increases. The goal of this paper is to provide insights and guidance to system designers and program managers, not necessarily familiar with cognitive processing, regarding the resource/performance tradeoffs, and to provide guidance on the costs and benefits of different approaches to cognitive processing. This paper is organized into two parts: the first introduces an analytical framework within which the relationships between task complexity and system complexity can be formulated; in the second part, we introduce and analyze a canonical architecture called context switching cognitive processing architecture that exploits heterogeneous and run-time reconfigurable processing hardware to address the conflict between operating range (i.e., breadth), and efficiency through dynamic specialization of processing capabilities to the current task demands.

15. SUBJECT TERMS
Cognitive function flow, Cognitive function in autonomous robotics, Binding of cognitive functions, Processing resources for cognitive algorithms, Neuromorphic processing for cognitive functions

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** THOMAS E. RENZ
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	47	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

TABLE OF CONTENTS

LIST OF FIGURES

1.0 SUMMARY

Unmanned Air Vehicle, (UAV) Autonomy constitutes a specialized domain of processing problems that demand computational architectures optimized to its needs. The unique combination of processing requirements stemming from the nature of the platform, the tasks to be performed, and the nature of the environment in which the systems operate requires the design of efficient and effective real-time cognitive processing architectures that can rapidly adapt to the demands of unpredictably changing tasks/environments. As processing resources are a limiting factor for autonomous operations in complex environments, the incorporation of new enabling low Size, Weight and Power, and Cost, (SWAP-C) processing technologies into autonomous systems is important to overcoming current limitations and keeping pace with peer adversaries. But with the increasing variety of processing technologies, the number of design choices for implementing end-to-end cognitive processing flows multiplies and the impact of these design decisions on efficiency and effectiveness increases. The goal of this paper is to provide insights and guidance to system designers and program managers, not necessarily familiar with cognitive processing, regarding the resource/performance tradeoffs, and to provide guidance on the costs and benefits of different approaches to cognitive processing.

Understanding the potential value of cognitive processing requires evaluating overall system costs and benefits of behavioral complexity. Typical analyses characterize computational costs of component functions independent of the end-to-end autonomous behaviors to which they contribute and the environments in which they operate. This type of analysis obscures key contributions that behavioral complexity can make to overall system efficiency and performance (e.g., energy consumption of executing complex trajectory planning algorithms needs to be weighed against the energetic gains of avoiding energetically expensive paths). It is important for designers to understand the impact of behavioral complexity beyond its computational costs and event to consider behavioral approaches to managing resource consumption and optimization. This paper is organized into two parts: the first part introduces an analytical framework within which the relationships between task complexity and system complexity can be formulated; in the second part, we introduce and analyze a canonical architecture called context switching cognitive processing architecture (CSCPA), that exploits heterogeneous and run-time reconfigurable processing hardware to addressing some of the key features and constraints of processing for autonomy.

We begin by introducing a framework for decomposing and characterizing both system and task complexity using a state space formalism. This formalism allows us to formulate complexity of the environment (i.e., physical systems) and complexity of the autonomous system (i.e., computational systems) within the same framework. The generality of the state-space formalism is equally important in our context in order to be able to analyze a wide variety of different computational models including both discrete and continuous systems, and asynchronous and synchronous system. Within the state space formalism we describe three distinct dimensions of complexity, Structure; Function; and Dynamics. Within this framework we relate task complexity to system complexity, and provide a framework for describing the different aspects of complexity of each and tradeoffs between them. In general, system complexity must be designed to match the requirements of task complexity, but there are a variety of different ways in which complexity can be distributed during system design. Of particular importance are the tradeoffs between specialization, configurability, efficiency, and autonomy. Mismatches between architectural optimizations and specific function/application characteristics result in inefficiencies. In

many application domains, specialization is a common design strategy in which behavioral breadth is traded for either greater efficiency (by exploiting structure of a simpler domain) or for higher performance (depth of processing in a particular domain). However, in autonomy, there is a tension between two key metrics for autonomy, efficiency and operating range. The current challenge of autonomy is to achieve a reasonable scaling between task breadth and system resources.

The conflict between operating range (i.e., breadth), efficiency, and performance can be addressed through dynamic specialization of processing capabilities to the current task demands to increase both autonomous performance and computational efficiency. This approach is enabled by run-time reconfiguration (RTR) architectures that allow hardware to change organization during the computation to tailor end-to-end processing chains as needed during different phases of the computation/behavior/task. Specialization of processing provides improvements in both the efficiency and performance of processing, over processing that is statically optimized to the global operational context. Context Switching Cognitive Processing Architectures are introduced as a canonical RTR architecture suited to autonomy applications. The core computational principle motivating context switched processing is the decomposition of complex task domains into piecewise simple domains, called contexts, enabling the use of lower complexity/specialized algorithms to achieve the task objectives within each domain. When operating in dynamic and unpredictable real-world scenarios, context sensitive run-time reconfiguration can temporally multiplex limited hardware resources. The potential benefit of run-time reconfiguration is the specialization of the context specific computation to the near-instantaneous needs of the task, reducing resources (e.g., the size and energy) required/consumed. These benefits of improved context specific processing performance/efficiency must be weighed against the costs of context monitoring and context switching (e.g., the additional space required to hold extra configuration information and the time and energy needed to reconfigure). One of the advantages offered by CSCPA is the ability to parallelize the Context Monitoring and the Context Specific Processing components. Our analysis supports the case for using heterogeneous reconfigurable processing hardware in implementing CSCPA when the number/complexity of contexts is sufficiently large, and the frequency of context changes is sufficiently low. We specifically considered the use of event based processing (e.g., TrueNorth) for context monitoring because of the low power needed for continuous monitoring of sparsely occurring complex events. The low precision, probabilistic, and approximate nature of event based processing techniques is well matched to the nature of contexts in the environment, which do not have precisely definable boundaries or features.

2.0 INTRODUCTION

In this paper we present a processing resource analysis for embedded systems that consist of heterogeneous processors. Different processing hardware architectures (General Purpose Processor, GPP, Graphical Processing Unit, GPU, Neuromorphic, Field Programmable Gate Array, FPGA, and Application Specific Integrated Circuit, ASIC) are efficient for performing different computational tasks. The overall performance of such a heterogeneous embedded system will depend upon the allocation of computational tasks to computational resources. The domain of interest in this paper is UAV autonomy which has a unique combination of processing requirements stemming from the nature of the platform, the tasks to be performed, and the nature of the environment in which the systems operate. The following attributes, in combination, shape the unique the requirements and constraints of processing for autonomy:

1. **Nature of the Task/System**:

a. **Closed Loop Control**: all autonomous behaviors inherently consist of closed loop interactions between the platform and the environment. The environment provides inputs to the platform sensors and the platform responds to inputs by acting upon the environment through actuators. The function of processing in autonomy is to optimally map sensor inputs to actuator outputs. The fundamental organizational structure of processing for autonomy consists of the control loop, consisting of elaborations of the basic sense-decide-act chain that is closed through the environment.

b. **Real-Time Control**: Typically there is a limited amount of time in which the autonomous system must respond to events in the environment. Most of the control tasks performed in autonomy therefore have latency requirements. These can often be very demanding (e.g., obstacle avoidance during high speed flight in cluttered environments), and optimization of end-to-end processing latency is often a key performance objective. Real-time, (RT) stream processing provides an appropriate algorithmic framework for the formulation of RT end-to-end processing chains for control.

c. **Performance criteria**: In many application domains, the primary objective of computation is to find exact or optimal solutions to a computational problem, while the minimization of computational resources is a secondary concern. In autonomy, whole system resource minimization (which includes computational resources) is often an equal or more important concern, and adequate solutions to computational problems that can be generated quickly and cheaply are often preferable to exact or near optimal solutions.

d. **Operating Range**: In many computational domains, the input and output spaces and the objectives of the computation are narrowly defined and known a priori, which leads to solutions that are deep (highly optimized for the specific domain) but not broad. One of the fundamental objectives of autonomy is breadth - the ability to operate in as wide a variety of environments as possible, and to execute as wide a variety of missions (objectives) as possible.

e. **SWAP-C**: The size, weight, power, and cost constraints on the autonomous system are typically considerably stricter than in other domains. These considerations not only affect the performance criteria as discussed above, but also limit the sensing, processing, and actuation capabilities of the system. These limitations affect the type, bandwidth, resolution, and quality of measurements of the environment, and affect the bandwidth, precision, and effectiveness of actions on the environment.

2. **Nature of the Environment**:

a. **Non-Stationarity**: Many of the algorithms designed today depend upon assumptions about the statistical stationary of the environment (e.g., linear time invariant systems) that allow the use of standard statistical estimation methods. Unless one artificially constrains the operating environment within narrow ranges, typical environments are highly non-stationary and frequently discontinuous. This strongly influences the nature of inputs to autonomous systems and the dynamic responses of the environment to system outputs.

b. **Information Sparseness**: the vast majority of sensor data is either predictable (due to the highly predictable nature of the environment) or irrelevant to the task being performed. The main occupation of processing in autonomy is the detection of and response to the sparse set of unpredictable and relevant events. Data may be unpredictable for several different reasons: truly stochastic behavior of the environment that cannot be predicted; insufficient sensing which does not provide enough resolution/accuracy/range of the variables needed for prediction; hidden or unobservable variables that influence the behavior of the environment; insufficient complexity in prediction models/or limited predictive processing resources that do not provide the needed accuracy/complexity to model the environmental phenomena of interest.

Computational, architectural, and algorithmic concerns, approaches, and solutions that conform to this unique collection of task, system, and environmental constraints and characteristics should therefore be considered to constitute a distinct sub-class of embedded computation, processing for autonomy, on a par with other classes of computation like cloud computing.

UAV autonomy requires the design of efficient and effective real-time cognitive processing flows. The use of a variety of specialized processing technologies provides unique advantages in addressing both the strict SWAP constraints on UAV processing and the demanding processing requirements of autonomous behavior in complex and rapidly changing environments. For example, event-based sensing and computation is an efficient alternative to traditional solutions, but it is not clear exactly for what platforms, tasks, and environments. As processing resources are a limiting factor for autonomous operations in complex environments, the incorporation of new enabling processing technologies into autonomous systems is important to overcoming current limitations and keeping pace with peer adversaries. But with the increasing variety of low SWAP-C processing technologies, the number of design choices for implementing end-to-end cognitive processing flows multiplies and the impact of these design decisions on efficiency and effectiveness increases [1]. The leading cause of delay between innovation and deployment in this arena is the combination of rapid evolution of new processing technologies, and the lack of tools to evaluate the resource and performance impacts of alternative end-to-end system design choices. In current design practice, performance/resource consumption assessment is limited to analysis and experimentation on specific implementations of individual component functions being executed on homogeneous processing hardware. Design of optimized system level autonomy requires the performance assessment of entire end-to-end flows, driven by and interacting with operationally realistic environments while executing on real heterogeneous processing platforms. Tools for evaluating the performance/efficiency of heterogeneous cognitive processing flows are becoming increasingly important for accelerating the transition of new processing technologies into autonomous systems. While resource requirements and performance of end-to-end autonomous systems have been evaluated for isolated system functions, integrated end-to-end cognitive processing flows have not received the same treatment. End-to-end cognitive systems require their own unique performance and resource metrics and analysis methods that differ significantly from those of their individual functional constituents. If clear metrics and methods existed for assessing costs and benefits of different end-to-end cognitive systems and alternative computing technologies, then system design and acquisition personnel could make systematic analyses and quantitative comparisons of alternative technologies leading to more informed decisions. The goal of this paper is to provide a reference document to be used as a decision-making aid to guide system designers and program managers not necessarily familiar with cognitive processing, or resource/performance tradeoffs of different approaches to cognitive processing, to provide guidance on the costs and benefits of different approaches to cognitive processing, paying particular attention to issues related to integration of cognitive process flows within both legacy and emerging UAVs and weapons systems. In the following we will address two topics: 1. Metrics appropriate for the design and analysis of end-to-end processing systems for autonomy. 2. The resource analysis of a particular canonical form for autonomous system processing.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Taxonomy and Metrics for Autonomy

One of the most challenging aspects of performing a quantitative comparison between end-to-end cognitive systems is their variety. In their most basic/abstracted form, end-to-end cognitive flows integrate sensor processing, decision making, and motor output, into a sensori-motor information processing loop that is closed through the environment which is both sensed and acted upon (Figure 1). The basic Sense-Decide-Act loop constitutes the basic structural construct of processing for autonomy. [2]

However, both in evolved and engineered systems, the elaboration of this basic pattern, and in particular the expansion of the intermediate processing between sensory input and motor output, has been associated with an increase in cognitive capability and behavioral flexibility. These systems can span a large range of complexity, capabilities, and operating ranges [3, 4]. In this section we will address metrics enabling analysis of a wide variety of full cognitive process flows.

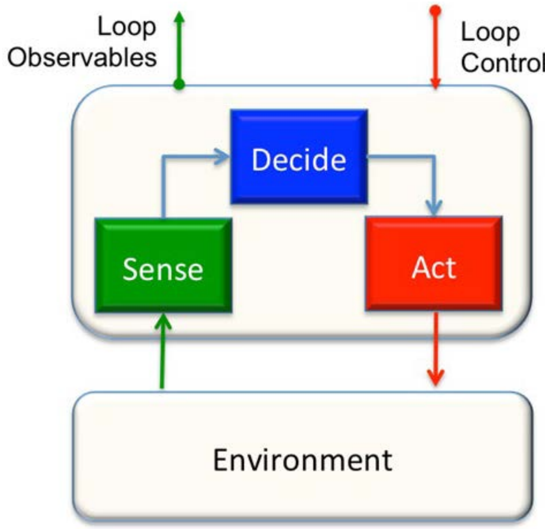


Figure 1: Illustration of Basic Sense-Decide-Act Loop

No single factor has a more ubiquitous effect on both performance and SWAP-C than system complexity. Additional complexity can always be exploited to achieve better task performance but there are complicated factors that affect the cost-benefit tradeoff of adding additional complexity. For example, evaluation of the value of current computations is complicated by their influence on future benefits/rewards, indicating the need for an amortized analysis (i.e., an analysis that takes into account not only instantaneous performance but also long term performance). A canonical example is the use of pro-active computation, in contrast with reactive computation. Reactive processing is always necessary in autonomous systems to cope with unexpected events in the environment. Pro-active processing, while adding complexity and not being strictly necessary, can however lead to significant performance improvements that can outweigh significant processing costs by predicting and preventing computationally or energetically expensive circumstances from arising (e.g., it is much more costly to perform an emergency maneuver to avoid a collision, than to predict it and avoid it long in advance).

It is therefore important to account for system complexity in analyzing cost/performance trends. Analysis of other factors influencing cost and performance will require that system complexity be factored out before making comparisons. In the following we will present a taxonomic breakdown of autonomous system complexity, both its descriptive components (which should be used to characterize the complexity and limitations of autonomous systems), and its prescriptive components (which should be used during design to determine the required system complexity). Taxonomies and metrics go hand in hand; the criteria used for splitting systems into groups must be based upon some form of metric. Therefore we will present an integrated discussion of taxonomies and metrics rather than discussing them separately.

3.1.1 Background.

AUTONOMY LEVELS FOR UNMANNED SYSTEMS (ALFUS) is the best known and most well established unmanned system (UMS) taxonomy [5]. It was a product of a cross-government ad hoc working group started in 2003 under the direction of the National Institute of Standards and Technology (NIST), Department of Homeland Security (DHS), and Army Research Laboratory (ARL), and published its results in 2008. Its goal was to develop a framework for characterization of autonomy for unmanned systems that provided standard terms and definitions for requirements analysis and specification, and metrics, processes, and tools for evaluation and measurement. The ALFUS metrics characterize autonomous systems along three dimensions: Mission complexity; environmental complexity; and human independence. These are a good set of metrics for characterizing key factors that are specified by external functional system requirements at the beginning of the system design process. However, these dimensions do not address the solution space.

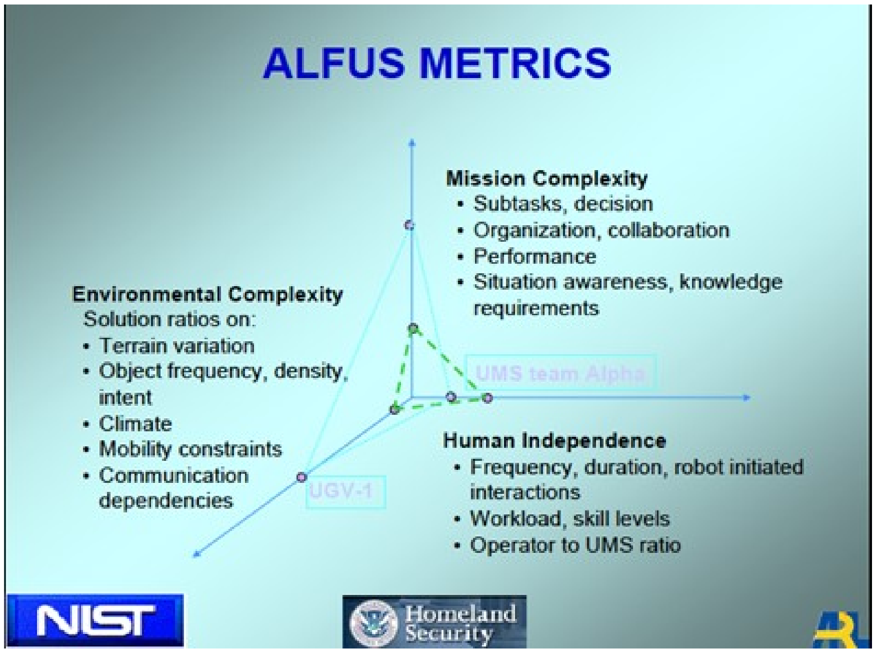


Figure 2: Illustration of the Three Primary Dimensions of the Autonomy Levels for Unmanned Systems Framework

While the ALFUS framework was a seminal study and many of the distinctions developed in that study are adopted here, this framework does not provide a means for characterizing internal system complexity needed in the design and comparison of alternative cognitive processing architectures, and the framework does not provide clear insights into the design tradeoffs between the different metrics. In the current work, we alter the ALFUS framework, combining some categories and splitting others to provide a framework useful from the prescriptive/engineering standpoint. The framework developed here captures key engineering decision points and requirements and dependencies between engineering decisions from top down perspective.

3.1.2 State-space Formulation of Complexity.

The complexity of systems, whether natural or man-made, can best be formalized using the concept of the system state space (also called phase space). The state space of a system is the space of possible configurations that the system can assume. For example, in statistical mechanics, the state of a gas is determined by the positions and velocities of all the particles in the gas. In a digital computer, the state would be the binary state of all the binary devices in the computer. However, in practice such fine grained analysis is not possible, and one must instead evaluate coarser macro-states of these systems. We will return to the topic of characterizing the complexity of the state space in the section on context switching systems. The generality of the state-space formalism is important in our context in order to be able to analyze a **wide variety of different computational models** including both discrete and continuous systems, and asynchronous and synchronous system. It also allows us to **formulate complexity of the environment (i.e., physical systems) and complexity of the autonomous system (i.e., computational systems) within the same framework**. It is conceptually useful to think of resource consumption as a measure of the resources needed to maintain states and to make state transitions, and to formulate processing latency in terms of the length of trajectories in state space.

Each of the system complexity measures we will be discussing (environmental, mission, system complexity) can be described along three distinct dimensions: **Structure**; **Function**; and **Dynamics**. Each can be conveniently formulated within the state space framework. All three types of characterization are useful for a full description of system complexity, but this triplet constitutes an over-complete description. Any two measures in the triplet will largely determine the third. For example, knowing the structure of the processing system and the dynamics of the execution of the processing determines the functional input-output relationship. In computational system design there is often a tradeoff between complexity in one of these dimensions and complexity in the others.

3.1.2.1 Structural Complexity.

Structural complexity captures the static/kinematic aspects of the system that define and constrain the possible states of the system and hence determine the complexity of the state space itself. Structural system characteristics correspond either to static aspects of the system that are fixed at design time, or to infrequently or slowly changing (in comparison with the timescales of the run time computations) characteristics of the system. It is largely determined by capabilities of fixed hardware, but can also include the constraints imposed by software, if the software is fixed (e.g., many FPGA implementations).

The structural complexity reflected in the state space is determined by the:

- **Dimensionality** or number of degrees of freedom (DoF) in the state space; Dimensionality and DoF can differ when the actual state space is constrained and consists of a subspace embedded in a space of higher dimensionality (e.g., strange attractors).
- **Geometry and Topology** of the state-space, defining neighborhoods of states, nearness of states, and distance metrics between states. State spaces with complex geometry and topology can be constructed with complex state transition functions where distance is defined in terms of number of transitions from one state to another. The structural complexity of an ASIC is captured in both the complex geometry/topology of its state-space and of the state-transition function which is determined by the particularities of the structure of a specific application.
- **Quantization** of the state-space, which includes both the dynamic range and the resolution of each of the dimensions of state space. Quantization need not be uniform, and may be specialized for the environment or task.

One can intuitively think of these factors as determining the **number of unique states that can be distinguished**. The state space is partitioned up into N dimensional volumes (cubes in the simplest case) and each distinct state is represented by a unique volume. The number of elements in the state space is often a key parameter in determining the computational complexity of algorithms. A critical task in the design of autonomous systems is specification of the state spaces for the environment, the mission, and the system, their dimensions, metrics, and quantizations.

SWAP-C

- **Cost**: Structural complexity is the dominant contributor to fixed/sunk costs in hardware determined at design time. Since structural complexity cannot be altered easily after the system is constructed, it is common for designers to engineer in excess structural complexity to create capacity to accommodate unanticipated future processing demands. It has been hypothesized that in biological systems, structural complexity comes at a low cost [6].
- **Power**: The idle power consumption is determined by structural complexity. Each active stateful device requires power to maintain state and be ready to respond. Another important contributor to power requirements is communication network needed for interactions between devices.
- **Size/Weight**: Greater structural complexity implies more devices which leads to larger and heavier systems. The scaling of auxiliary systems such as power supplies may dominate the scaling of size and weight as structural complexity increases.

3.1.2.1 Functional Complexity.

Functional complexity describes the complexity of the input-output mapping or the objective function being optimized, without reference to the implementation of the mapping. This is only relevant to systems in which there are subspaces of the state space identified as input dimensions (e.g., sensor state variables) and output dimensions (e.g., actuator state variables). Functional complexity captures the complexity of externally observable mapping performed by the system, which can be assessed based upon a black-box analysis. Characteristics of mappings such as number of input/outputs, number of dependencies between inputs/outputs, non-linearity, non-convexity, discontinuity, and asymmetry contribute to their complexity. Figure 3 shows a system with low functional complexity.

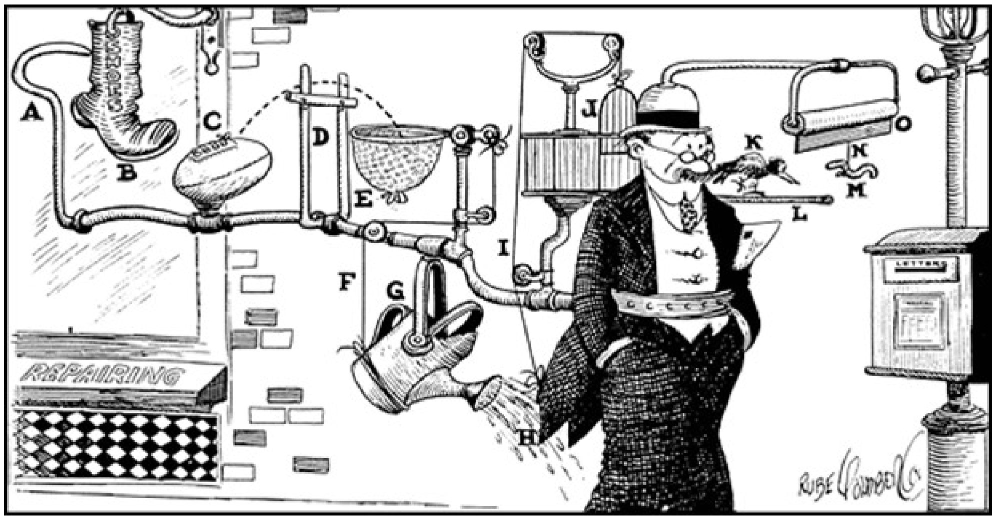


Figure 3: Rube Goldberg Machines Have Low Functional Complexity but High Structural and Dynamic Complexity

For engineered/designed systems, the input-output mapping is specified by functional requirements of the task to be performed. The input-output mapping can be described either:

- Explicitly with an exhaustive, exemplar based, or sample based set of input-output pairs, or with equations; or
- Implicitly with an objective function or reward function and constraints whose optimal solution is the desired mapping

In autonomous systems, it is most common to specify functional requirements implicitly as an optimization problem.

The concept of function can also be applied to natural/non-engineered systems (e.g., the environment) in a well-defined way by considering the governing equations of the system as their input-output function. The governing equations provide a mapping from current observable state of the system to next states. For example the behavior of many physical systems can be derived from the optimization of an objective function (energy function/Hamiltonian). This mapping may depend upon unobservable variables.

The Vapnik-Chervonenkis (VC) dimension offers a possible computational complexity measure with which to quantify Functional complexity. This is a measure of the ability of the functional map to distinguish and respond to different situations differently. It is a measure of behavioral flexibility.

SWAP-C:

Since functional complexity is by definition independent of implementation, it is difficult to establish anything but theoretical lower bounds on the processing complexity and SWAP. It does however have a **profound effect on the overall SWAP of the autonomous system** (i.e., the plant). For example, the energy consumption of navigating to a target will depend critically on the navigation behavior executed, and whether the platform can intelligently anticipate and avoid time and energy consuming situations. The behaviors executed by the autonomous system are determined by the input-output mapping. If we are discussing a control system (as all autonomous systems essentially are), the input-output map constitutes the control law. One can evaluate and compare different control laws using metrics such as energetic efficiency, robustness, speed of convergence, etc. These are key behavioral performance measures for the system as a whole, rather than just for processing. Increasing functional complexity, while it may incur a computational cost, may be critical for reducing overall system SWAP. The greater functional complexity achievable by a larger platform (due to its ability to incorporate more processing resources) may provide behavioral energetic advantages that offset the larger processing SWAP-C. Conversely, in order for a larger SWAP platform to be operationally useful / feasible, it may be necessary to give it greater functional complexity and hence processing SWAP-C in order to offset its overall SWAP-C.

3.1.2.1 Dynamical Complexity.

Dynamic/Behavioral complexity describes the complexity of state space trajectories as the system evolves in time. The system dynamics are specified by: a state transition function that explicitly describes the mapping of the state at one instant to the state at the next instant (e.g., a program); differential or difference equations (e.g., a dynamical system); or by the implicit optimization of an objective function (e.g., minimization of an energy function or other trajectory characteristics such as curvature or length). Figure 4 shows a system with high dynamical complexity.

In the resource analysis of end-to-end system cognitive processing, measures of interest are defined on end-to-end state trajectories. There are many ways to characterize the complexity of state space trajectories. For systems in which there are distinguished input and output subspaces, the trajectories of interest are typically those corresponding to the propagation of information from input (changes in state of input subspace) to output (changes in output subspace). One simple measure is the length of the trajectory, which is a measure of the latency of the end-to-end computation. The calculation of the energy consumed during end-to-end computations can be partitioned into energy consumed in maintaining state and energy consumed in state transitions. The total transition energy consumed in an end-to-end computation is the path integral of the state-transition energy function. In physical systems, the complexity of state space trajectories can be descriptively characterized by the rate at which new information is generated. For simple deterministic phenomena, the trajectory complexity is low due to the predictability of the next state. Information generation rate is an important measure of the complexity of dynamics. As discussed in greater detail in the section on reconfigurable computing, temporal multiplexing of hardware leads to more complex computational dynamics in both system activation and configuration state-spaces.

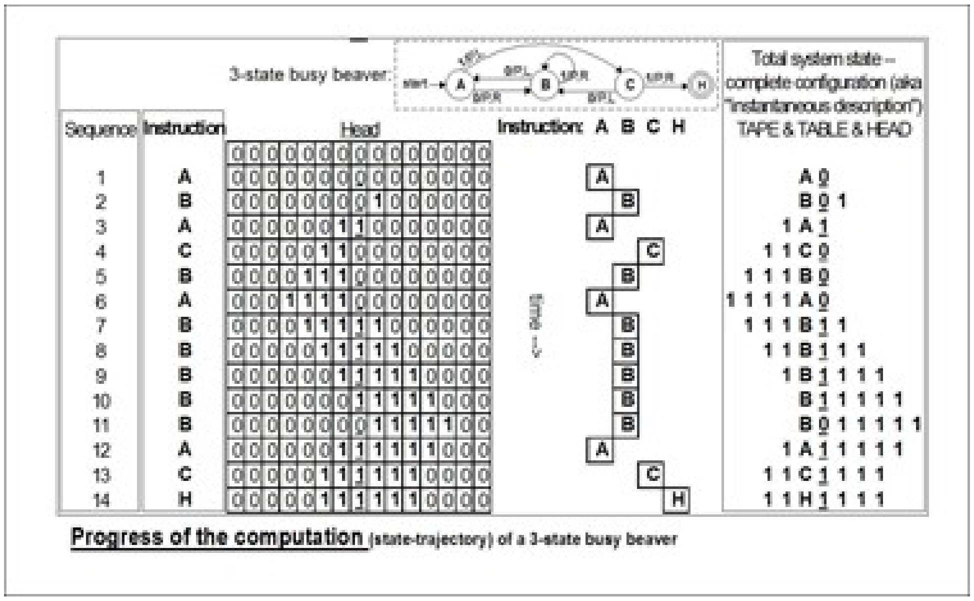

3-state busy beaver:
start
A
B
C
H
Total system state – complete configuration (aka "instantaneous description") TAPE & TABLE & HEAD
Sequence
Instruction
Head
Instruction: A B C H
time -->
Progress of the computation (state-trajectory) of a 3-state busy beaver

Figure 4: Busy Beaver Turing Machines Have Low Structural and Functional Complexity, but High Dynamic Complexity

The following examples illustrate the mapping from common computational concepts to characterizations of computational dynamics:

- Parallelism in processing is reflected in the state-space trajectories that move in multiple state-space dimensions simultaneously.
- In synchronous processing the trajectories progress in regular discrete time steps, while in asynchronous processing, the trajectories can progress in continuous time or irregular discrete steps.

SWAP-C:

Each active change in state of a processing system consumes time and energy. In general, larger changes in state will consume more time and energy. Motion of the state in different dimensions of the state space will incur different energy and time costs. In particular it is important to note that in processing systems, there can be significant dynamics in both the activation state space (corresponding to the execution of "within context" computations) and in the configuration state space (corresponding to context switching). The costs associated with state transitions during program execution will be significantly different from the costs of system reconfiguration that occurs during context switching.

3.1.3 Complexity of Autonomy.

There are dual aspects of the complexity of autonomous systems: one aspect captures the task complexity and determines the requirements imposed upon the designed system, constituting the independent variables over which the designer has no control (although see below under mission complexity for caveats); the dual aspect is the system complexity, which includes many aspects of the system other than processing (although we will be focusing only on sensing, processing, and actuation system components), and is the result of engineering design decisions. Figure 5 shows that system complexity must be commensurate with task complexity. Growth of system complexity with respect to task complexity (right) is the main challenge of autonomy.

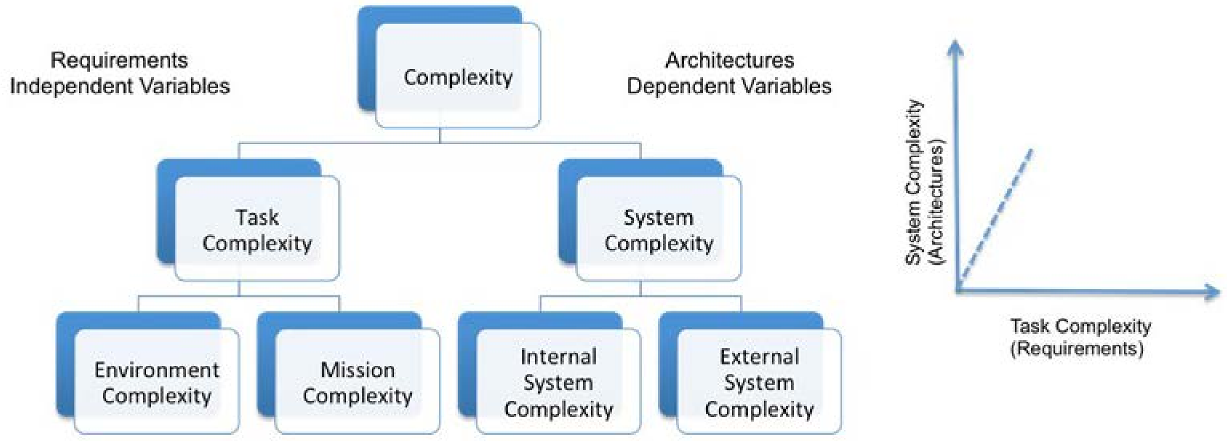


Figure 5: Taxonomy of Complexity for Autonomy

3.1.3.1 Task Complexity.

Task complexity and performance requirements place a lower bound on the complexity of the autonomous system that can execute the task at the desired level of performance. Task complexity combines two of the dimensions in the ALFUS framework: Environmental complexity; and Mission complexity. While these are typically beyond the control of the system designer and come in the form of system requirements, there is an important feedback from the designer during system design, in which the designer alters the design requirements by specifying system operating constraints. These specify both limitations on the environment complexity (e.g., cannot fly in winds stronger than x) and on the mission complexity (e.g., can only track up to x simultaneous targets).

Environmental Complexity

The environment or range of environments in which the autonomous system must operate will have a significant impact on the computational burden on the system and is an important factor in determining the requisite system complexity. For example, the processing load associated with navigating in a sparse environment differs significantly from the processing load associated with navigating in a heavily cluttered environment. As discussed in the introduction, two characteristics of the environment that have a significant impact on the processing load are: stationarity/non-stationarity; and predictability/stochasticity/information generation rate/rate of innovation.

- ***State Space Structure*****:** The state of the environment can be described at many different levels of granularity. In practice, the appropriate level of description is largely determined by the mission requirements and by the behavioral resolution of the system. The finest scale at which the system can sense and act upon the environment sets a lower limit on the relevant resolution of the environment state space. The dimensions of the environment state-space are not typically the same as for the sensor state space. For example, for a navigation task, the relevant environmental state space could be the 3D positions and motions of all the objects in the environment, but the sensors may not be able to detect these features directly. The maximum precision with which positions and velocities in the environment can be inferred will be determined by the sensors. The dimensions of the environmental state space need not correspond to physical dimensions, nor do they need to be tied to an absolute coordinate system. They can be conceptual (e.g., object categories) and relational dimensions (e.g., relative speed). Relevant environmental states can include unobservable/hidden variables such as the internal states of other systems in the environment (e.g., internal state of an evasive target) that can affect their observable behavior.
- ***State Space Function*****:** The governing equations of the dynamics of the environment can be expressed in a variety of different forms. The trajectory of a thrown ball is governed by Newton's laws but also can be expressed as the minimization of an objective function (Hamiltonian formulation). In the case of animate objects in the environment (e.g., other aircraft, people, machines) their behavior can also be described by governing equations that can often be expressed as the optimization of an objective function that captures intent or purpose. It is important to note that governing equations can vary with respect to the state of the environment, and it is not uncommon to have large abrupt changes in the governing equations. This gives rise to situations in which different regions of the environmental state space exhibit very different dynamics (non-ergodic/non-stationary behavior).
- ***State Space Dynamics*****:** The dynamics of the environment is a product of both the state of the environment and the governing equations (structure and function). Dynamics can be described either implicitly as the solution of the equations of motion or state transition function, or explicitly as a trajectory in state space. Even if the current state is known with certainty, the trajectory can be stochastic due to the stochasticity of the equations of motion/transition function. The complexity of the dynamics of the environment determines how predictable the environment is, which in turn has a significant impact on processing load.

Mission Complexity

- ***State Space Structure*****:** Mission state consists of the state of autonomous system with respect to state of the environment, both points being in the state space of the environment (e.g., if the mission is to navigate to a target, the current state is the position and heading of the UAV relative to the target). The description of the autonomous system state does not include any internal system state variables that are not part of the environment state space.
- ***State Space Function*****:** Mission objectives can be expressed in a number of ways: by defining a set of goal states (environment state, autonomous system state) pairs; or by defining an objective function on the set of all pairs that is to be optimized.
- ***State Space Dynamics*****:** Mission dynamics are the product of both the dynamics of the environment and the dynamics of the autonomous system. Mission objectives can change as a function of both of these states. Changes to mission objectives can have a significant impact on both processing and overall system resource demands.

3.1.3.2 System Complexity.

Usually the designer tries to find the simplest system architecture that will satisfy the task requirements. SWAP-C is proportional to complexity since more complex systems usually consist of more components, more complex components, and more expensive components. There are however interesting tradeoffs that can be made between the complexity of the non-processing system components and the processing components in which: one can make do with less complex non-processing components by increasing behavioral/processing complexity (e.g., can maintain temperature either with an internal temperature control system, or behaviorally - by going into the sun when it is cold); likewise, one can make do with simpler processing if more complex non-processing elements are used (e.g., elastic properties of materials can be exploited to passively conserve energy).

In order to account for systems that are semi-autonomous and/or are part of a larger distributed system, we broadly divide system complexity into internal system complexity, and external system complexity. This distinction is important, as many systems shift complexity from internal systems to external systems as a means for lowering the SWAP-C in exchange for sacrificing full autonomy. It has been proposed that a measure of autonomy is the ratio of the algorithmic complexity of the internal system and the communication complexity between the internal system and external systems [7]. Figure 6 illustrates both the notional tradeoff between internal and external system complexity with respect to autonomy, and the splitting of task complexity between internal system complexity and external system complexity. The split determines the degree of autonomy of the system.

While the ALFUS framework specifically refers to human independence as a dimension for characterizing autonomous systems, we use a more general framework in which external systems can be either human or machine. External systems such as larger platforms, ground stations, networks/clouds, supercomputers, or humans can provide a resource constrained system with additional resources as long as the systems have sufficient bandwidth to communicate the needed information. While communication with external systems is desirable in that it can provide useful information/resources that are not available locally, the dependence on external systems introduces an undesirable vulnerability that can be exploited by adversaries, (e.g., denial of communications or manipulation).

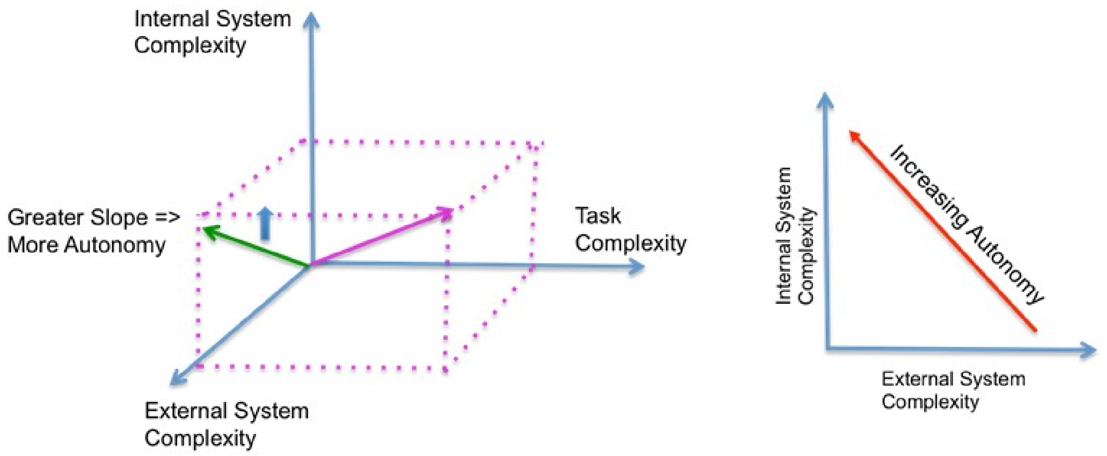

Internal System Complexity
Greater Slope => More Autonomy
Task Complexity
External System Complexity
Internal System Complexity
Increasing Autonomy
External System Complexity

Figure 6: Illustration of the Growth of System Complexity with Task Complexity

System designs with adjustable autonomy should be considered a compromise that permits resource savings by depending on external systems when communications are available and can move toward progressively greater autonomy as communications degrade. The price for this compromise is that such systems must be designed to include enough “margin” to ensure that processing resources are available when increased autonomy is required. This has the side effect of requiring more SWAP-C to accommodate the additional processing resources, which will consume additional power, fuel, etc. even when the additional processing resources are idle.

Internal System Complexity

In processing systems it is convenient to distinguish two different types of system state, **configuration state** and **activation state,** and to factorize the state-space into the product of the **configuration state-space** and the **activation state-space**.

- Configuration state corresponds to the values of all the adjustable settings of the system that determine the system operation (e.g., programs). Configuration changes include loading new programs, such as during reflashing of an FPGA/neuromorphic processor, or context switching in a GPP. Configuration state is particularly important when we consider learning and adaptation. Configuration parameters include slowly changing global mode parameters such as energy saving modes and clock speeds, to operating system parameters governing things like memory allocations and priorities, to very fast changing context switching and configuration of control logic during program execution.
- Activation state corresponds to the dynamic state of the system that changes as data is being processed. In digital computation, the activation state includes values of variables in memory, stacks, caches, registers etc. In neuromorphic processing, the activation state includes the activation values of each of the neurons, synapses, etc. A program execution can be viewed as a trajectory in activation state-space.

Figure 7 shows the decomposition of internal system complexity into activation and configuration complexity. These are primarily distinguished by timescale, but are conventionally associated with data and control/instruction processing. The complexity of any state space can be decomposed into structural, functional, and dynamical complexity. Structural complexity is determined by degrees of freedom, quantization resolution, and geometry and topology of the state space. In formal terms, the main distinction between these is in the timescales on which they change (slow and fast respectively – more generally one should consider a spectrum of timescales). Informally, the distinction is between control/instruction and data. Computational architectures often include separate communications paths for data and for control signals for reconfiguration. The independent consideration of these two subspaces is only valid when the behavior in configuration space is independent of the behavior in activation space. In general this will not hold. In neural systems an important component of the activation state is the membrane voltage of neurons, whereas an important component of the configuration state would be the strengths of synaptic connections between neurons.

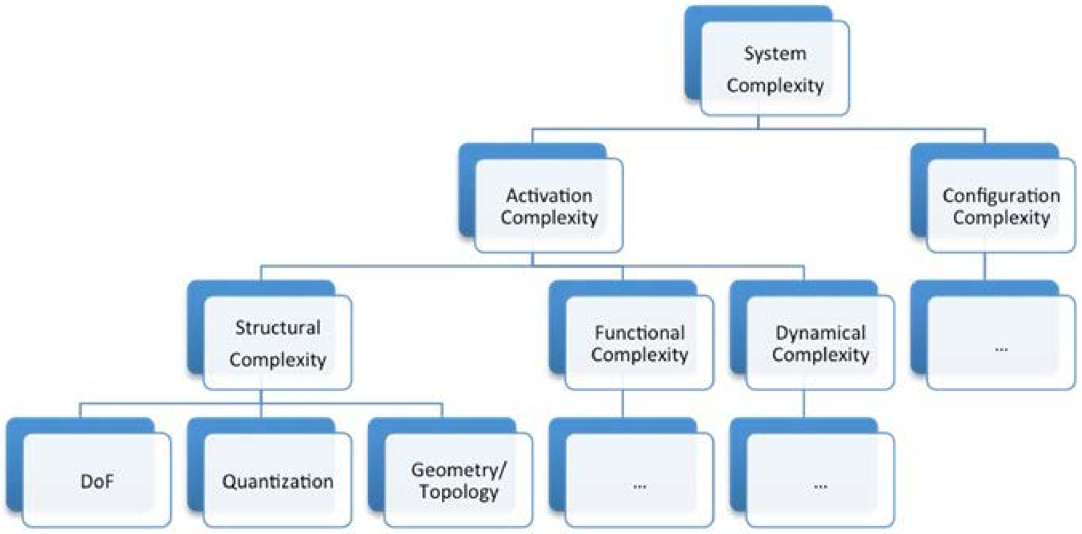


Figure 7: Decomposition of Internal System Complexity into Activation and Configuration Complexity

It is common in processing system design to trade off structural complexity and dynamical complexity to implement a given function (simple system doing more work vs. complex system doing less work) [8]. A comparison of complex instruction set computing (CISC) architectures that build complex instructions directly into the hardware and reduced instruction set computing (RISC) architectures that can accomplish the same things using simpler hardware but at the cost of more/complex activity serves as a well-known example of this tradeoff [9]. By constructing an activation state space in which relevant states are close or well organized, one can simplify dynamics. The distinction between software and hardware reflects the differences between dynamics and structure. For example, one can implement the same algorithm on an ASIC or in a GPP. The ASIC has high structural complexity (e.g., many irregularities/asymmetries in the circuit design leading to complex state space geometry and topology) and low dynamical complexity (e.g., no dynamics in the activation state space associated with program execution; no configuration state space dynamics). The central processing unit (CPU) implementation has low structural complexity (e.g., simpler state space geometry and topology), and high dynamical complexity (e.g., complex dynamics due to program execution). To be scalable, processing hardware design depends a great deal on structural regularity/symmetry. This gives rise to regularity/symmetry in the geometry and topology of the state space. These regularities give rise to low complexity according to the definition of algorithmic complexity. As task complexity (ie. functional complexity) increases the system complexity must also increase, but there is a design choice to be made about how one splits that additional complexity between structural and dynamic (Figure 8).

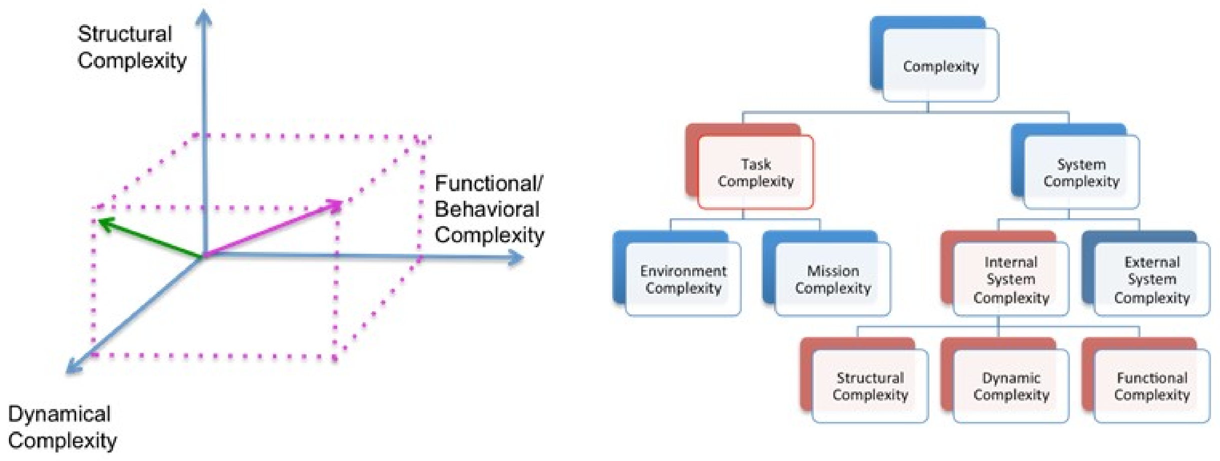


Figure 8: Relationship Between Functional or Task Complexity and System (Structural and Dynamic) Complexity

As part of the design tradeoff between structure and dynamics important design decisions must be made regarding the allocation of complexity to the activation state space and to the complexity of the configuration state space. This is also a tradeoff in which, greater complexity in the configuration state space/dynamics can result in simplification of the activation state space/dynamics,([see discussion of context dependent processing).

Supporting dynamic reconfiguration has costs and benefits. Creating complex activation state spaces without configurability imposes a large design time cost [8]. Creating complex activation state spaces with configurability imposes a run time cost. Comparing an ASIC, an FPGA, and a GPP illustrates this point. An ASIC has few degrees of freedom in its configuration space – its configuration is fixed at design time to support a particular application – but has complex (application specific) activation state-space geometry/topology and dynamics; an FPGA has many degrees of freedom in its configuration space but it is costly (in time and energy) to reconfigure and the structure of the activation state space is less complex; a GPP has a relatively few degrees of freedom in configuration space, but highly complex configuration space dynamics (which results in time and energy costs).

- State Space Structure: Internal system state can be decomposed into activation and configuration subspaces. From the hardware perspective, the state space consists of the possible states of the hardware. The actual state space may be considerably more constrained, consisting only of the set of states that can be reached from initial states using valid transition functions. As with the state-space of the environment, the relevant granularity of the internal system state space will depend on the function being performed. Characterization of structural complexity will leverage a key architectural design pattern for incrementally expanding capabilities of cognitive systems through the addition of new sensori-motor loops on top of existing sensori-motor loops.
- State Space Function: The function of the internal system is characterized in terms of the mapping performed from state in the input subspace, to state in the output subspace. This function can be described either extensionally in terms of input, output state pairs, or intensionally with equations or objective functions.
- State Space Dynamics: Internal system dynamics is described by the trajectory of system states that are determined by the state transition function.

3.1.3.3 Specialists, Generalists, and Autonomy.

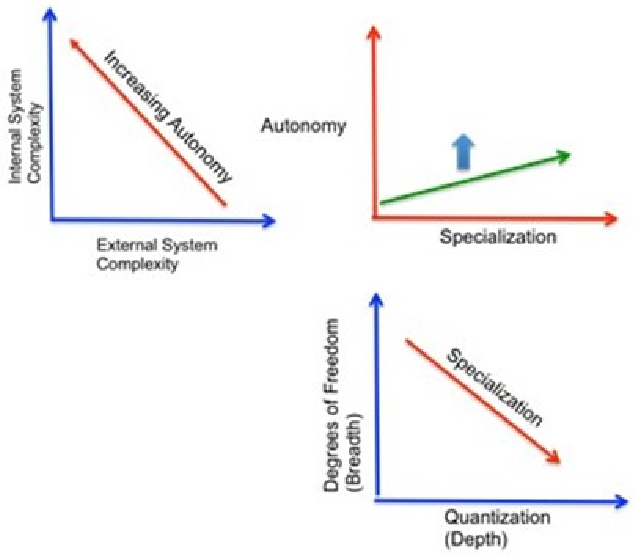
In general, the more task complexity is reduced, the greater the degree of autonomy we can achieve with limited resources [10, 11]. Specialization is a common design strategy in which behavioral breadth is traded for depth in order to achieve a performance objective within given resource constraints. Constraining the task can introduce regularities that can be exploited to reduce required system complexity. Task specialization is the restriction of either the operating range/environment of the system or a restriction of the breadth of the mission. At some point, if the task is made simple enough, we cross the line from the autonomous into the automatic. The current challenge of autonomy is to achieve a reasonable scaling between task breadth and system resources. It is important when comparing different autonomous systems, to compare systems of similar degree of specialization. Matching task complexity to resources, or resources to task complexity, is an important part of the design process.

Specialization can be understood as a reduction in/redistribution of the state-space complexity of the system enabled by a corresponding reduction in the state-space complexity of the task/environment. System state space complexity can be reduced through a reduction in dimensionality/degrees of freedom, complexity of quantization, or geometric/topological complexity, Figure 9). For example, by operating within a restricted environment, environmental complexity will be reduced as a result of reducing the number of degrees of freedom and increasing the predictability of the environment (e.g., operating in an environment with a constant temperature). The corresponding system state-space can also be of reduced dimensionality (e.g., no need to monitor or regulate temperature). There are a variety of different ways in which environmental complexity can be reduced for the autonomous platform:

- External/User restriction of operating range
- Internal/Behavioral restriction of operating range: the platform autonomously avoids certain environmental conditions so that it stays within valid operating range.
- Active Control of the Environment: the platform acts on the environment to maintain a restricted set of states.
- Partitioning environments into parts that can be handled autonomously and parts that require the help of external systems (e.g., maybe autonomous platform can cope with navigating around objects that don't move, but need help navigating around moving objects).
- Gross/Coarse quantization/control of some state space dimensions and fine quantization/control of others.

With the consequent reduction in task complexity, systems can either achieve the same performance with fewer resources, or can achieve higher performance with same amount of resources.

An important example of the latter strat-



Internal System Complexity
Increasing Autonomy
External System Complexity
Autonomy
Specialization
Degrees of Freedom (Breadth)
Specialization
Quantization (Depth)

Figure 9: The Relationships Between Internal and External Complexity in the Context of Autonomy; Between DoF and Quantization in the Context of Specialization; Between Specialization and Autonomy

egy involves the reduction in state space dimensionality and the increase in the quantization resolution of some of the remaining dimensions, so that the overall number of states remains approximately constant. This increase in resolution on fewer dimensions will permit the system to implement more precise and accurate behaviors.

3.1.3.3 Behavioral approaches to regulation of computational load and optimization of system resources.

Typically one thinks of system behavior as being governed by processing/computational systems for the purpose of achieving mission goals. There is, however, a class of behaviors, the purpose of which is to govern processing/computational systems themselves. This class of behaviors can be considered to be a subset of homeostatic behaviors that are intended to control/maintain internal state. There are two types of homeostatic mechanism: internal/covert regulatory mechanisms; and external/overt mechanisms. In the case of regulation of processing, an internal regulatory mechanism might be to throttle inputs to match the throughput of processors and avoid buffer overflows (e.g., increasing the threshold on event based sensors to produce fewer events). An external or behavioral regulatory mechanism might be a UAV slowing down flight speed in cluttered environments to maintain a constant sensor data generation rate. [12, 13] These behaviors are similar to those mentioned in the previous section devoted to regulating the environment in which the system operates, and within our framework, has the effect of simplifying the complexity of the state space dynamics of the environment. In contrast, the internal regulatory mechanisms have the effect of reducing the complexity of system state-space dynamics. Figure 10 shows a UAV performing a navigation/collision avoidance task. At slow speeds, the rate at which data needs to be processed and responded to are lower (top) than at high speeds (bottom). Behavioral regulation of information rates (e.g., by regulating speed) is an important technique for managing data with resource constrained computing.

While these behaviors may seem secondary to those directed at accomplishing mission goals, when resources are limited and maintaining autonomy is sufficiently important, these behaviors can be of equal importance to accomplishing mission goals.

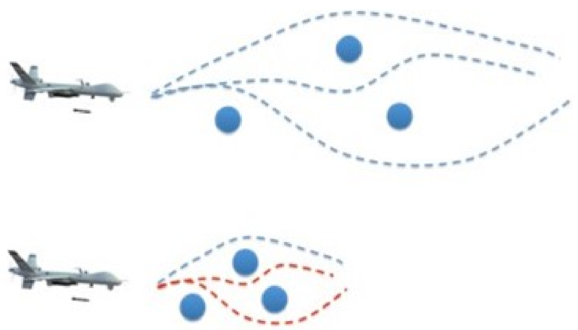


Figure 10: UAV Achieving a Collision Avoidance Decision

3.2 Resource Analysis of Cognitive Reconfigurable Computing

The cost structure for processing in autonomy applications differs from other classes of processing task [7, 14]. Key metrics for autonomy, efficiency and operating range are addressed by architectures that can accommodate real-time reconfiguration, tailoring end-to-end processing chains to the current demands of the environment and task. Traditional metrics of optimality, precision, and accuracy become soft constraints rather than hard requirements in autonomy applications. As we enter new era in computing technologies (the “era of dark silicon”) in which we can place more transistors on a silicon die than we can afford to turn on at their maximum operating speed, energy efficiency determines performance, and the energy efficiency of reconfigurable architectures may be their key asset [15].

In this section we present a particular canonical form for cognitive processing architectures, which we call Context Switching Cognitive Processing Architectures (CSCPA). As will be discussed in greater detail below, from the hardware standpoint this architecture belongs to the class of run-time reconfigurable (RTR) computing architectures [16]. From the control systems/functional standpoint this approach belongs to the class of Hybrid control systems. We will argue that context switching architectures are: well suited to the nature of processing for autonomy; provide a convenient form for analysis; and are general enough to accommodate a wide variety of autonomous processing systems. We believe this class is canonical in the sense that: 1) completeness/computability/sufficiency: any autonomous control algorithm can be either implemented or approximated to any desired accuracy with an algorithm from this solution space; and 2) complexity/optimality: there is a control algorithm in the space of solutions which has a complexity that approximates to any desired accuracy the complexity of an arbitrary optimal algorithm.

3.2.1 Background.

One of the most important design degrees of freedom in processing architectures is the degree of programmability/configurability. The more programmable the architecture (i.e., degrees of freedom in system configuration space), the greater the variety of functions it can perform. As processors become more specialized/less configurable, they become faster and more efficient by exploiting function specific architectural optimizations. Mismatches between architectural optimizations and function/application characteristics, result in inefficiencies. Designers have chosen to balance this flexibility-efficiency tradeoff in different ways resulting in a wide variety of different processing architectures. Many processor technologies are defined by where they fall within the configurability/efficiency space (e.g., FPGA) [16].

At one extreme of programmability, universal machines, any computable function can be implemented by programming the device after it has been created (i.e., post-fabrication programming). The von Neumann general purpose processing (GPP) architecture heavily shares (e.g., through temporal multiplexing) a single or small number of generic compute elements which are rapidly and frequently reconfigured using instruction bits to perform a specific task (i.e., high dynamic complexity of configuration state). Re-configurability does have costs. Holding programs and reconfiguring functionality comes at the cost of area: area to store the configuration; area for gates that have more functionality than strictly necessary; and area for wires that may not be used. In cases that are not fully spatial (e.g., stored-program processors), we also pay for energy-reading configurations from memory. These costs result in lower performance, higher area, and higher energy than a fixed-function component. But these costs can be amortized over a large set of applications and users and over the lifetime of the device. For these reasons, general purpose von Neumann architectures gained a foothold early on in the development of processing architectures [8, 16].

At the other extreme of programmability, application specific integrated circuits (ASICs), a large number of compute elements is used without multiplexing, each of which performs a single dedicated operation during a computation. For typical dedicated computing applications the designer attempts to tailor the organization of the machine to a particular application or algorithm, or even a particular data set, so as to maximize performance, minimize area, and minimize energy [8].

Spatially distributed processing elements and efficient interconnection networks are configured to exploit application specific data and instruction locality characteristics resulting in reduction in computation time and computation energy. Inefficiencies occur when there is a mismatch between architecture parameters and application characteristics such as: locality (as measured by the Rent Exponent, i.e., p_arch vs. p_app); and word width (i.e., task size vs. device component size).

Between these extremes there are devices whose configurability is limited spatially, temporally, and/or functionally (e.g., FPGAs, GPUs, Digital Signal Processors, DSPs, Coarse Grained Reconfigurable Architectures, CGRAs). Even within the realm of universal machines there are a range of processor architectures that fall on the spectrum between few general purpose multiplexed computational elements (e.g., RISC) to many special purpose less multiplexed computational elements (e.g., CISC). Reconfigurable computing (RC) addresses performing computations with spatially programmable architectures (e.g., FPGAs). A key differentiator in reconfigurable processing architectures is whether the reconfigurable resources are controlled with a static configuration, like FPGAs, or with multi-context memories (e.g., Very Long Instruction Word, VLIW-style CGRAs) [17]. To distinguish cases where the configuration remains constant during an application from dynamic/on-line reconfiguration, the latter is termed run-time reconfiguration (RTR). RTR allows for hardware to change organization during the computation as needed during different phases of the computation/behavior/task [10].

There are many organizational scales on which reconfiguration can occur, ranging from local gate level to whole system and multi-system levels. The individual processor architectures discussed above each constitute optimizations to achieve a particular tradeoff between speed, energy, area, and flexibility, often built to support a single set of parameters tuned to the homogeneous characteristics of particular applications. In practice, end-to-end applications do not have homogeneous characteristics, but rather contain a mix of sub-computations with different characteristics. The famous 90/10 rule from Knuth suggests that 90% of the runtime is spent in only 10% of the code. Such a profile might benefit from a hybrid architecture that has two components: one that focused on area minimization for the 90% of the code that runs only 10% of the time; and another focused on maximizing computational density and minimizing the energy for the 10% of the code that runs 90% of the time [18]. At the board level, a variety of processor architectures supply building blocks from which larger scale heterogeneous processing architectures can be constructed. This has led to designs that combine the area efficiency of a GPP with the computational density and energy efficiency of a spatially reconfigurable compute engine (e.g., commercial processor-FPGA hybrids such as Xilinx Zynq devices combining ARM cores and FPGAs). In the following we will discuss a run-time reconfigurable heterogeneous processing architecture that is targeted at autonomy applications in which dynamic configuration of end-to-end processing flows will benefit performance both by enabling dynamic specialization of processing capabilities to the current task demands to increase both autonomous performance and computational efficiency.

3.2.2 Context Switching Cognitive Processing Architectures.

The core computational principle motivating context sensitive processing is the decomposition of complex task domains into piecewise simple domains, called contexts, enabling the use of lower complexity/specialized algorithms to achieve the task objectives within each domain. This is similar to piecewise approximation of functions. By choosing a sufficiently fine decomposition of the domain of the function, linear approximations to the function will be valid within each domain element. With coarser partitions of the domain, more complex (e.g., quadratic) approximations will be needed to keep error low. In essence, context switching factorizes complex algorithms into conditionally independent components (conditions are contexts), concentrating the computational task of condition testing into a separate dedicated processing component. The architectural components of the context switching computational architecture are:

- **Context monitoring**: this computational component continually monitors information from sensors (of both internal and external state) and inputs from other sources (e.g., external systems, top down inputs) to detect changes in the context, and when a context change occurs, to characterize, recognize/classify the new context.
- **Context switching**: once a new context is recognized, this component will recall/derive a processing configuration specific to the new context, and will reconfigure the end-to-end cognitive processing flows governing autonomous system behavior. The configurations will consist of: assignments of programs to processors; settings of program and processor parameters; and setting of information flows both within and between processors.
- **Context specific processing**: this computational component constitutes the end-to-end cognitive processing that provides the control systems needed for operating within the current context.

Figure 11 illustrates these components in a possible instantiation of a context switching architecture in which contexts are linked to processing configurations via a lookup table. Context monitoring, switching, and specific processing are all executing in parallel on hardware resources. Right shows an instantiation of the architecture in which an array of context monitors continuously receives data and evaluates in parallel. Triggering of a particular context causes an associated context specific program to be loaded into context specific processing module.

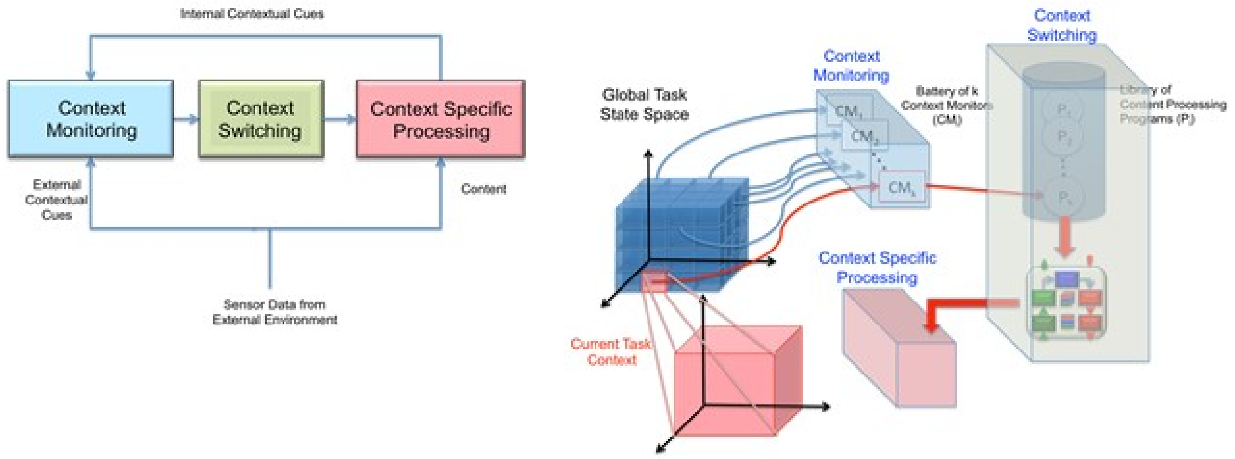


Figure 11: Diagram of Functional Components of a Context Switching Processing Architecture

Contexts and Complexity

The CSCPA is well suited to the computational demands of autonomous operations in natural environments because natural environments tend to be composed of non-uniformly distributed local niches with discontinuous boundaries and locally stationary statistical characteristics. The local/stationary structure of the context can be exploited to simplify processing within that context. These contexts are encountered discontinuously and unpredictably during autonomous operations. As the task state leaves one context and enters another, the autonomous processing changes to suit the new context. This approach, which can be broadly included in the class of divide and conquer computational methods, has been widely applied in different application domains (e.g., gain scheduling in flight control). Within each context, the simplification of the state-space allows for specialized/special purpose/context specific processing, resulting either in a reduction of computational resources needed, and/or an increase in the performance achievable within the context. Specialization of processing to the local operating environment provides improvements in both the efficiency and performance of processing, over processing that is statically optimized to the global operational context.

With a fixed set of hardware resources, scaling autonomy to more complex tasks requires that the resources be used more efficiently. When operating in dynamic and unpredictable real-world scenarios, context sensitive run-time reconfiguration (RTR) can temporally multiplex limited hardware resources. The potential benefit of run-time reconfiguration is the specialization of the context specific computation to the near-instantaneous needs of the task, reducing resources (e.g., the size and energy) required/consumed. These benefits of improved context specific processing performance/efficiency must be weighed against the costs of context monitoring and context switching (e.g., the additional space required to hold extra configuration information and the time and energy needed reconfigure).

A unique design aspect of this type of architecture is the need to co-optimize the definition of contexts and the context specific processing. Contexts are portions of the task (environment and mission) state space. Examples of broad contexts are:

- Environmental Contexts: flight regimes (e.g., wind conditions), environmental complexity (e.g., urban/rural, high clutter/low clutter)
- Internal Contexts: state of resources (e.g., battery level, weapons, processor loading), system health, physical state (location, velocity), behavioral state
- Mission Contexts: threat level, op-tempo (e.g., high speed/low speed), mission phase (e.g., explore/seek, track/pursue, exploit/attack/consume)

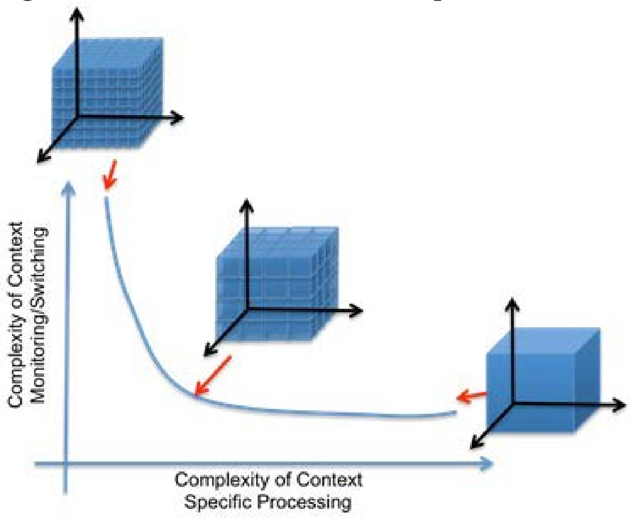
It is far more common in autonomy for the environment to be the trigger for either a change in function being performed, or a change in the way in which a particular function is being performed. We will refer to this situation as data-driven reconfiguration in contrast with task-driven reconfiguration.

Partitioning of the environment into distinct contexts is determined both by the inherent characteristics of the environment (e.g., statistical characteristics, ergodic decomposition) and by the ability to discriminate different contexts, both from the standpoint of detection/characterization and action (i.e., if two distinct contexts do not have different context specific processing associated with them, then there is no reason to treat them as separate). The granularity with which contexts are defined (i.e., quantization of state space into contexts), will have a significant impact on computational costs, where the computational costs are the sum of the costs of context monitoring, context switching, and context specific processing. There will be a tradeoff between complexity of context monitoring/switching and complexity of context specific processing. Figure 12 shows of the tradeoff between the granularity of the partitioning into contexts and the complexity of the context specific processing. The notional graph shows three different granularities of partitioning of the state space ranging from the degenerate single element partition on the lower right, to a very fine partition at the upper left, corresponding to different context monitoring/switching complexities.

There is some freedom in creating a context partition on the part of the designer, who needs to balance the costs of context monitoring and switching with the costs of context specific processing. At one extreme, one large (degenerate) context can be created which never changes, in which case there are no costs associated with context monitoring or switching, and all the complexity needed to cope with complex environments must be contained within the context specific processing. At the other extreme, a context partition can be created with many elements that are so small that, within each of these contexts the environment can be considered constant so that the context specific processing for each context may be as simple as constant functions. In this case the computational cost of context specific processing is minimized, and most of the complexity (and cost) is pushed into the context monitoring and context switching components. In this case, monitoring for many small contexts is expensive, and context switching will occur frequently, incurring a context switching cost. Another cost that needs to be accounted for is the design cost of implementing a context specific algorithm for each context in the partition. If the partitioning is very fine and there is no systematic way of deriving context specific algorithms from context characteristics, the burden of designing individual context specific algorithms will be untenable.

Here we conjecture an analogue to the 90/10 rule, for the domain of autonomy:

90% of the computational resources of autonomous systems are spent in executing the 10% of the code (context specific processing).


Complexity of Context Monitoring/Switching
Complexity of Context Specific Processing
Figure 12: The Tradeoff between Granularity of the Partitioning into Contexts and Complexity of the Context Specific Processing

This is saying that autonomous systems will spend most of their time on context specific processing, but need to be continuously monitoring and respond quickly to changes in context that will occur sparsely and unpredictably. The other 90% of the code that is dedicated to context monitoring and switching must be implemented so as to minimize the computational resource consumption. By creating a context partition at the appropriate level of granularity, the CSCPA will efficiently handle the 90/10 split in autonomy tasks by: exploiting context specific structure to optimize the efficiency of context specific processing; minimizing the effort needed for context monitoring and the frequency of context switching (i.e., see discussion of efficient event based context monitoring below).

We also conjecture that another form of the 90/10 rule holds for autonomy:

90% of the runtime in autonomous systems is spent in monitoring for 10% of information that is task relevant (requiring a response).

3.2.3 Analysis of CSCPA Performance.

There are many factors to consider when designing an autonomous system to perform a task. All aspects of the processing, detection and even system idling need consideration when evaluating processing architectures. This work aims to describe an example analysis process to follow when evaluating designs with respect to overall power usage.

For this analysis, we will use the standard definition of power, where W is power measured in Watts, J is energy measured in Joules, and t is time in seconds:

$$W = \frac{J}{t} \tag{1}$$

The goal is to minimize the value of J by finding the optimal combination of the processors, algorithms, and overall logic.

3.2.3.1 Example Scenario.

Imagine designing a system for a UAV flying a reconnaissance mission. We are tasked to design an autonomous control system that can adapt the control law to the current weather conditions and terrain. If the drone encounters a significant change in environment, such as the change between flying in a dessert and then flying near mountains, it will adjust its flight control, (Figure 13). This system is power constrained, and the mission specifies that the task must complete in a defined amount of time.

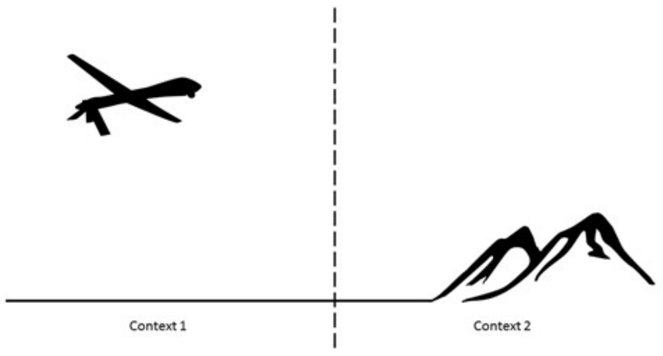


Figure 13: A Drone Flying in Two Separate Contexts, Which in this Case are Environments

3.2.3.2 Problem Breakdown.

We will break this problem into three main parts: context monitoring, context switching, and context specific processing.

Context Monitoring

This component of the system has the job of monitoring the environment and triggering any changes necessary. This can potentially be done in two different manners: a polling approach, or an event-based/interrupt approach [19].

In a polling approach, we imagine a sensor that is constantly sampling its environment at some frequency, (Figure 14). With each poll, a small amount of processing is performed that determines if a reconfiguration is needed, such as a change in UAV navigation or sensing algorithms. During the rest of the time, the polling system lays idle, waiting until its next scheduled time to poll once again.

In an event-based/interrupt system, there is no continual checking needed. Rather, the system relies on asynchronous signals/interrupts to trigger a context switch. These signals can come from sensors or other systems at any time. Upon receipt of trigger signals, associated reconfiguration can be performed immediately.

Events triggering reconfiguration can range in complexity from very simple (single bit changes) to very complex (complex configuration of events).

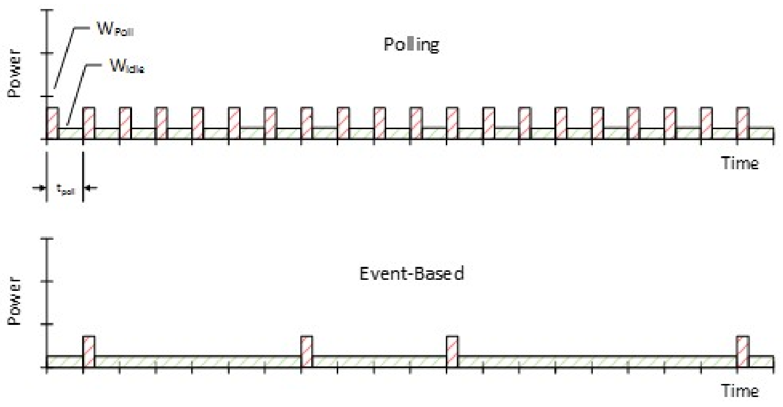


Figure 14: Comparison of Polling and Event Based Sensing Schemes

Context Switching

Context switching refers to a reconfiguration process that may consist of multiple subtasks, such as saving previous state, recalling context specific algorithms, loading in new or saved parameters, and initialization [20]. Each of these aspects take a certain amount of time and power to complete, so they too must be considered when trying to minimize the overall power consumption of the system, (Figure 15).

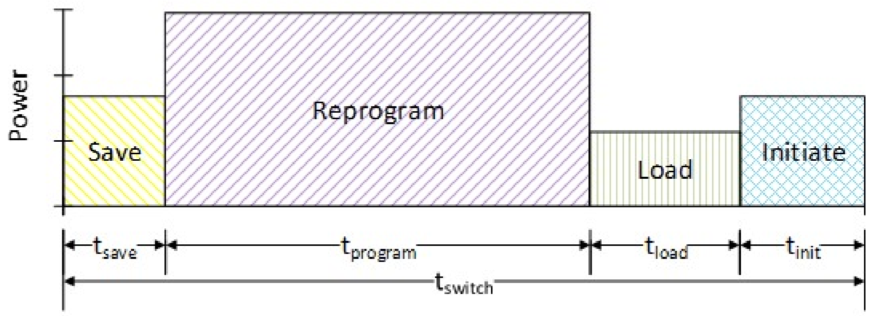


Figure 15: Example of a Context Switch Timeline

Context Specific Processing

The last and likely most significant consumer of power for the overall system is the context specific processing that is running continuously between context switches. In the UAV example, this would be the navigation and flight control algorithms. Depending on their complexity, this processing may be very power hungry and time consuming. Algorithms for different contexts may differ in complexity. If the system often uses the more expensive algorithms, it will consume more power. The overall complexity will therefore depend upon the probability distribution across contexts. Figure 16 shows an example of this. There are periodically context changes that trigger different algorithms. Each algorithm has its own power consumption.

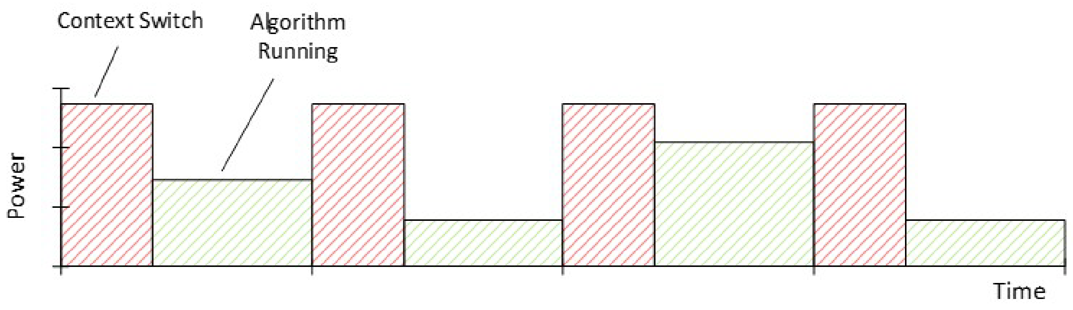


Figure 16: Example of Algorithm Timeline

3.2.2.3 Estimating Power Consumption.

In this paper, we will consider four different processors types: Central Processing Units (CPUs) [21, 22], Graphics Processing Units (GPUs) [23], Field Programmable Gate Arrays (FPGAs) [24, 25, 17], and neuromorphic processors like IBM's TrueNorth [26, 27]. The same process can be considered for nearly any processor type in order to get a rough estimate of which architecture would work best for the task.

Context Monitoring

In order to estimate the amount of power that the polling or event-based scheme might consume, we first estimate how often we are going to be polling for information, or how often we are going to be receiving events. We will define the polling scenario frequency as f_{poll}, measured in Hz. The polling rate will determine the amount of power necessary. This estimate includes the cost of reading sensor data as well as the cost of processing this data to determine if a context switch condition has occurred. After obtaining and processing the data, the monitoring system will sit at idle, waiting for the next scheduled time to poll, (Figure 17).

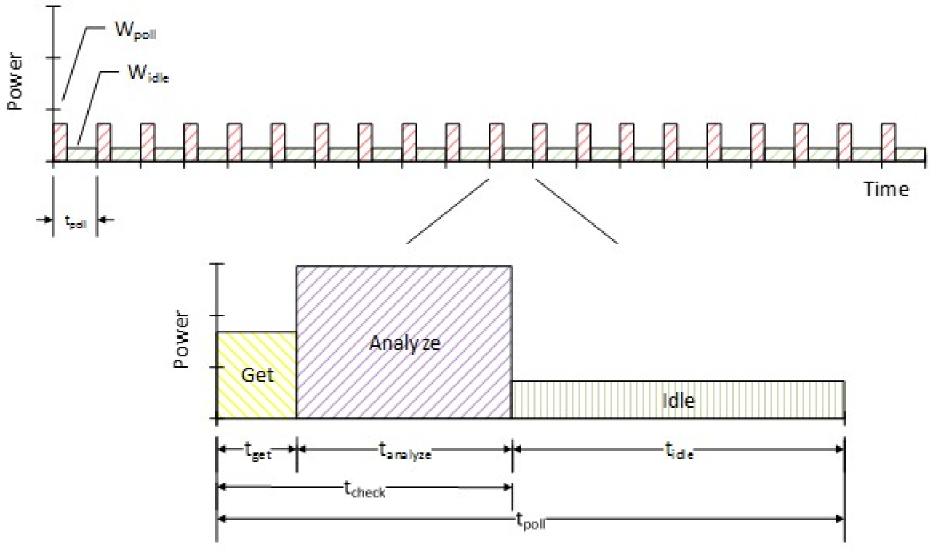


Figure 17: Breakdown of Polling Scheme

The average power consumed by polling is defined as W_{check}

$$W_{check} = W_{get} + W_{analyze} \tag{2}$$

Applying Equation 1, the values of J_{get} and $J_{analyze}$ need to be determined, as well as their respective time values. These values are determined by the algorithms and the data that is needed during each polling event. If the sensor is an infrared camera looking at the heat signature from the ground, then the “get” process would require the retrieval of an entire image frame, and the processing might involve analysis of the whole frame. In contrast, if the sensor is a thermometer, reading and processing a single temperature value then the cost will be dramatically less.

Using estimates of the power consumption per operation and the approximate number of operations per polling operation (#op), we can estimate the number of Joules necessary to perform a poll.

$$J_{get} = \#_{op,get} J_{op} \tag{2}$$

$$J_{analyze} = \#_{op,analyze} J_{op} \tag{3}$$

Similarly, we can determine the amount of time for each component of polling.

$$t_{get} = \#_{op,get}\, t_{op} \tag{4}$$

$$t_{analyze} = \#_{op,analyze}\, t_{op} \tag{5}$$

Finally, the amount of time and power used during the idle time is easily calculated as follows:

$$t_{idle} = \frac{1}{f_{poll}} - \left(t_{get} + t_{analyze}\right) \tag{6}$$

The idle power can obtained from processor datasheets when available, or can be derived from performance benchmarks, otherwise. We estimate that the idle power consumed is approximately 5% of the designed thermal design power (TDP) of a processor, where TDP is the approximate amount of thermal power dissipated by processor under normal operation. While this is not an exact measure of processors power consumption, it is often a good estimate. Typically this (TDP) value for CPUs and GPUs can be obtained from spec sheets.

We estimate the energy of the idling systems as follows:

$$J_{idle} = W_{idle} t_{idle} \tag{7}$$

Combining all the power values, we obtain the poll power required.

$$W_{poll} = f_{poll}\left(J_{idle} + J_{get} + J_{analyze}\right) \tag{8}$$

Event-based polling can be similarly analyzed. Because event-based polling is asynchronous, we must use an estimate of the average frequency of events in place of a fixed polling frequency. We call this estimated event frequency f_{event}. This can be used identically as f_{poll}.

Context Switching

In the estimation of power consumption used in context switching, we separate the switching process into 4 different portions, each of which can be analyzed separately. These four segments are: Save state; Load Program; Reprogram; and Initialize. Each of these segments serves its own purpose and may function slightly differently depending on the platform on which they are running. However, the final result will still be an estimation of the average amount of power necessary, (Figure 18).

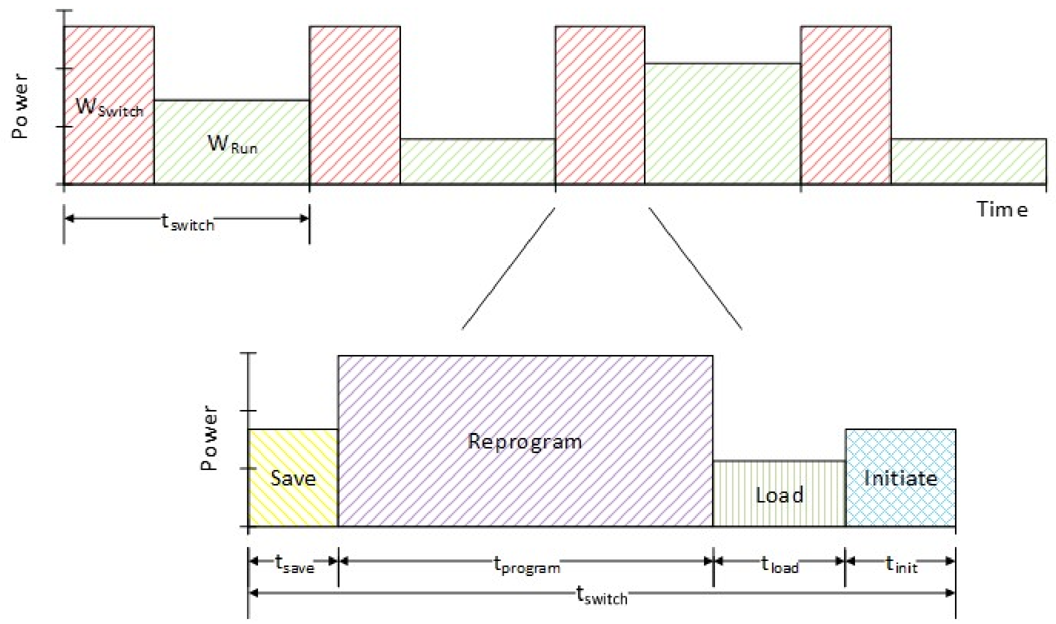


Figure 18: Breakdown of Context Switching Element of the System

As with the context monitoring, we can estimate the amount of power necessary to perform context switches as:

$$J_{switch} = J_{save} + J_{program} + J_{load} + J_{init} \tag{9}$$

The save state phase includes all necessary saving of weights, results, and other values that need to be stored from the previously run algorithm. This could also include the saving the program execution state that may be needed to reinstate processing later. This may not always be necessary or even possible, depending on the algorithm and processor type, but is must be considered when it is.

The amount of power consumed in switching context is highly dependent on the amount of information that needs to be stored. If a large table needs to be kept, it will likely require more power than a few cached weights. We estimate the required power using values from processor specification sheets. If the amount of data to be stored is known, or can be projected, it is simply as follows:

$$J_{byte,w} = \frac{W_{mem}}{BW_{write}} \tag{10}$$

$$J_{save} = J_{byte,w} \#_{byte,w} \tag{11}$$

The value of $J_{byte,w}$ represents the amount of power necessary to write one byte of memory, when W_{mem} is the rated power of the memory being used, and BW_{write} is the write bandwidth of that same memory. The total power needed to save is then simply the amount of number of bytes needing to be stored multiplied by the cost of each byte, as seen in Equation 12. Finally, the amount of time this procedure will take is found by taking $\#_{byte,w}$ and dividing by the write bandwidth of the memory.

The amount of power needed to reprogram the processor is the most difficult to estimation. This will vary greatly between processor types. Reprogramming a CPU and GPU will be fast due to their design for temporally multiplexing hardware. However, for an FPGA or a TrueNorth, where they are more designed to be single purpose, loading a new algorithm can likely mean re-programming/re-flashing the chip entirely[27, 17].

As a conservative measure for making sure enough power is supplied, it is best to assume that the reprogramming process is going to consume near the max amount of power that the processor is rated for. For a CPU or GPU, this value would be listed as the TDP from a specification sheet. For an FPGA, it would be safe to estimate this value by using the maximum voltage and current values from its voltage regulator. Using the power formula, $P = I \times V$, we can get an estimate of maximum power used. It is not very likely that this much power will be used, but an overestimation will ensure enough power is available. Lastly, for something like IBM's True-North, since published data is scarce, we can just take the highest recorded power values and time for our accepted programming power.

While finding exact values for the programming of a specific chip is difficult, once a value is acquired, it can be applied to the same formulas as everything else to give the amount of power required to reprogram in an algorithm.

$$J_{program} = W_{max} t_{program} \tag{12}$$

By multiplying the estimated power, by the approximate about of time necessary to reprogram, $t_{program}$, we can get the number of Joules required.

The next portion requiring estimation is the loading of weights, tables, and other data that may need to be loaded for the algorithm to be run. This is the inverse of the saving portion of the switching element, and can thus be estimated in much the same way. The only true difference is that the amount of power needed is not determined by the write bandwidth, but by the read bandwidth. Using the read bandwidth we get the equations for loading to be as follows:

$$J_{byte,r} = \frac{W_{mem}}{BW_{read}} \tag{13}$$

$$J_{load} = J_{byte,r} \#_{byte,r} \tag{14}$$

Finally, we must consider the possibility of the system needing to initialize some other components. For some systems, this may not be necessary, but for others it might. This includes the need to send messages, open up I/O ports, and whatever other sort of initialization needs to take place. Because this process will vary highly, we will use the same general estimation seen in the analyze phase of the sensing element, presented above. This initialization can be generalized as an algorithm, much like what the above described analysis is. Making this assumption, the amount of power necessary is simply the number of operations necessary multiplied the average power per operation.

$$J_{init} = \#_{op,init} J_{op} \tag{15}$$

With all elements of the switching stage estimated, we can finally complete the equation for the power required for switching.

$$W_{switch} = \frac{J_{save} + J_{program} + J_{load} + J_{init}}{t_{switch}} \tag{16}$$

$$t_{switch} = t_{save} + t_{program} + t_{load} + t_{init} \tag{17}$$

3.2.2.4 Analysis.

The final component of the process is evaluating the amount of power that is needed to run the algorithm that is controlling the system overall. For the drone example, it could be navigation system that is keeping the drone flying and not colliding with anything in the environment. This portion of the system is highly dependent on the algorithm of choice, but nonetheless can still be put into a general form. As with the initialization phase of the context switch and the analyze phase of the polling scheme, we can make assumptions to break it down into a simple calculation of the number of operations multiplied by the estimated amount of power per operation.

$$J_{alg} = J_{op}\#_{op,alg} + J_{byte,w}\#_{byte,w} + J_{byte,r}\#_{byte,r} \tag{18}$$

$$t_{alg} = \#_{op,alg}t_{op} + \# \tag{19}$$

$$W_{alg} = \frac{J_{alg}}{t_{alg}} \tag{20}$$

It is also very likely that the algorithm will take require sort of memory reading and writing, so we can use the same estimations that we saw in the save and load portions of the context switching. These values can then be added to the operation costs of the algorithm itself to finally give the value of J_{alg}. Using the value of J_{alg} and an estimated time to complete the algorithm, we can get a value for W_{alg}.

The above calculation of W_{alg} only assumes that the algorithm is going to be run once through, but it is going to run for a while, assuming that the context the plan is flying through does not change after one single iteration of the algorithm. Because of this, we need to assume some sort of frequency of condition changes that would cause a context switch, denoted by f_{change}. With this assumption, we can calculate how much power, on average, the entire algorithm and context switching will take. The inverse of f_{switch} gives the amount of time between context switches, and since we already have the amount of time the switch will take, we can calculate the time that the algorithm will run, as well as the total amount of joules it will require.

$$t_{analysis} = \frac{1}{f_{change}} - t_{switch} \tag{21}$$

$$J_{analysis} = W_{alg}t_{analysis} \tag{22}$$

Now that the value of $J_{analysis}$ has been found, we can combine it with the value of J_{switch} to get the average power of the entire algorithm processor, including context switches, at some given frequency of change, f_{change}.

$$W_{analysis} = (J_{switch} + W_{alg}t_{alg})f_{change} \tag{23}$$

Combined Estimations

With the calculations of the power for the polling, switching, and analysis setups of our systems, we can combine all of them to get an approximate value for the amount of power that the system will require to run constantly, and then see how much power something like a battery might need to supply to keep it running.

$$W_{system} = W_{analysis} + W_{poll} \tag{24}$$

With the value of W_{system}, you can estimate the amount of power necessary over any given amount of time on average.

$$J_{total} = W_{system}t_{total} \tag{25}$$

These final values of J_{total} and W_{system} can then be used to estimate the amount of power, and even the voltage and current required to keep the system running.

Real-World Estimation Context Monitoring

Nearly all of the estimation above is assuming generalized information, but it is important to understand how this process would likely happen with commercial off-the-shelf (COTS) products. For this section, we will outline this process with popular processors in all the above mentioned categories: CPU, GPU, FPGA, and TrueNorth, (Table 1).

Table 1: General Information for COTS Processors Used in Evaluation of Respective Power Consumption

Processor	Architecture	Manufacturer	Model	Idle Power (W)	TDP	Clock Speed
CPU	x86	Intel	Core i7 2600K	4.7500	95.0000	3.4 GHz
TrueNorth	True North	IBM	TrueNorth	0.0040	0.0730	1 kHz
FPGA	FPGA	Xilinx	Virtex5	1.5000	N/A	100 MHz
GPU	GPU	Sapphire	Radeon HD 7970	15.0000	300.0000	925 MHz

As described in the sensing calculation section, we need to obtain several values in order to determine an estimated power for the polling or event-based scheme: J_{get}, $J_{analyze}$, t_{get}, and $t_{analyze}$. For these calculations we can use the estimated clock speed that is typically found on a specification, or the clock speed that is defined when writing the FPGA program. This also, may be difficult to retrieve for a neuromorphic processor, like the TrueNorth, but a rough order of magnitude estimate will yield relatively similar estimates to the actual values. Using this clock speed, and the TDP, we can estimate the number of joules per operation, by dividing TDP by the clock speed.

$$J_{op} = \frac{TDP}{Clock\ Speed\ \times\ \#_{cores}} \tag{26}$$

Next, we need to get the number of operations for the poll or event and then the analysis. This part is very dependent on the situation, and thus can only loosely be demonstrated. For this, we will assume that the polling takes 100 operations and the analysis takes approximately 5000 operations. This is very naïve, as it is not likely the process will be identical for each processor type, but for simplicity sake, we will assume that they will be. With the number of operations, we can figure out how quickly this can happen, by dividing the number of operations by the quantity of the clock speed multiplied by the number of threads on the processor. This assumes that the process can be easily parallelized. This may not be true, so if it is not, simply ignore the number of threads, which makes it as if the process is completely single threaded.

$$t = \frac{\#_{op}}{Clock\ Speed\ \times\ \#_{cores}} \tag{27}$$

Using these values and estimates, we calculate and estimate the necessary power for each processor type to complete a poll. Along with this and a frequency of polling, which we will assume is once every 10 seconds or 0.1 Hz, we get an average power to poll the environment for each type.

As seen in Table 2, because the operations require so few cycles, the amount of power consumption comes mainly from the system idling. However, if the polling and analysis were to be more intensive and require more operations, this total power value would increase.

Table 2: Estimated power values for each processor type

Processor Type	Model	Clock Speed	TDP (W)	# threads	W_{poll} (W)
CPU	Core i7 2600K	3.4 GHz	95	1	4.75001354
TrueNorth	TrueNorth	1 kHz	0.073	256	0.04087273
FPGA	Virtex5	100 MHz	30	4	1.50015109
GPU	Radeon HD 7970	925 MHz	300	512	15.0001652

Real-World Estimation Context Switching

The same process can be followed for the rest of the system to get their respective power estimates. We will follow a similar process for the switching portion. For this analysis, we will ignore the "init" portion of the process, under the assumption that no new initialization will be necessary, but this could easily be included if it is actually required.

For all of the memory portions, the saving and loading of parameters, we will also assume the same information is being saved in all scenarios, and the same form of memory is being used. Because there is no difference in our memory type or the amount of information being saved or loaded, we can just assume a single value for this. For example, both J_{save} and J_{load}, can be assumed to be approximately 0.1 Joules each, and will take approximately 1 millisecond. We make this assumption for brevity, but if the memory type were to vary for each processor type, we would calculate the cost of each memory operation and the number of memory operations necessary thus giving us the total cost of the saving and loading data.

The most costly portion of the context switching is the actual reprogramming of the processor. Much like the idling that we see in the sensing portion, we must make some broad assumptions regarding the cost of reprogramming a processor. For a CPU and GPU, which are more designed to do many different tasks, the switching is relatively cheap, but for an FPGA and IBM TrueNorth, these systems likely need to be re-flashed completely, which takes more time and energy. For all systems we will assume that the power required is equal to the TDP, but the amount of time necessary will vary. We will make the assumptions that are listed in the Table 3.

As seen in Table 3, we assume that the TrueNorth and FPGA take much longer to program. Because of this elongated programming time, the necessary power to program is sometimes higher than with the CPU or GPU despite the TrueNorth and FPGA having a significantly lower TDP. While this will not matter all that much if we are switching contexts infrequently, but must be considered more if contexts need to be changed more often.

Table 3: Estimates of the Power Required to Program the Processor to Run Algorithms

Processor Type	Model	TDP (W)	$t_{program}$ (sec)	$J_{program}$ (J)
CPU	Core i7 2600K	95	0.001	0.095
TrueNorth	TrueNorth	0.073	5	0.365
FPGA	Virtex5	30	0.5	15
GPU	Radeon HD 7970	300	0.001	0.3

With the estimated cost of the programming the processor, as well as our assumption about the cost to save and load parameters, we can figure out the energy necessary for the context switch. Table 4 combines all of these values. These final values will be used after we evaluate the cost of the actual algorithm running.

Table 4: Estimated Costs of the Different Portions of Context Switching

Processor Type	Model	TDP (W)	t_{prog}	J_{prog}	$J_{save/load}$	$t_{save/load}$
CPU	Core i7 2600K	95	0.001	0.095	0.1	0.001
TrueNorth	TrueNorth	0.073	5	0.365	0.1	0.001
FPGA	Virtex5	30	0.5	15	0.1	0.001
GPU	Radeon HD 7970	300	0.001	0.3	0.1	0.001

Real-World Estimation Context Specific Processing

The final part to consider is the algorithm that we are going to be running. This algorithm is running constantly, as long as a context switch is not occurring. This is very similar to how we treated the idle power during our estimation of the sensing power. Because this will be running constantly, we need to figure out on average how long between context switches, as these are the points that terminate and begin an algorithm running. For this example estimation, we will assume that this process will take place every 100 seconds.

We will assume that there are no read or write operations in the algorithms, for simplicity. Another assumption is that all potential algorithms will take on average 500k operations to complete. Using these assumptions we can figure out how long this will take and the amount of energy, (Table 5).

Table 5: Estimates of Power to Run the Algorithm Constantly Given 500k Operations are Necessary

Processor Type	Model	Cores	Threads	t_{alg} (s)	W_{alg} (W)
CPU	Core i7 2600K	4	2	7.35E-05	47.5
TrueNorth	TrueNorth	256	128	3.91	0.0365
FPGA	Virtex5	16	8	6.25E-04	15
GPU	Radeon HD 7970	2048	128	4.22E-06	18.75

Using the values in Tables 3-5, and the process described in the analysis section of Estimation of Power Consumption subsection, we can get the amount of power necessary to run the entire analysis (Table 6).

Table 6: Estimated Values of the Cost to Run the Whole Analysis

Processor Type	Model	t_{switch} (s)	J_{switch} (J)	W_{alg} (W)	t_{alg} (s)	$W_{analysis}$ (W)
CPU	Core i7 2600K	0.003	0.295	47.5	99.997	47.501525
TrueNorth	TrueNorth	5.002	0.565	0.0365	94.998	0.04032427
FPGA	Virtex5	0.502	15.2	15	99.498	15.0767
GPU	Radeon HD 7970	0.003	0.5	18.75	99.997	18.7544375

Final Total Power Estimation

Now that everything is estimated we can combine the sensing scheme with the analysis to get an estimate of the amount of power to run the entire system. This can be any combination of the different processor types. A CPU could be running the polling while a TrueNorth is running the analysis. We simply need to add the two power values together to get the total approximate amount of power to run everything.

3.2.2.5 Potential Realizations of the CSCPA.

There are a variety of ways in which the CSCPA can be realized using current technologies. One of the advantages offered by CSCPA is the ability to parallelize the Context Monitoring and the Context Specific Processing components. Our analysis supports the case for using heterogeneous reconfigurable processing hardware in implementing CSCPA when the number/complexity of contexts is sufficiently large, and the frequency of context changes is sufficiently low.

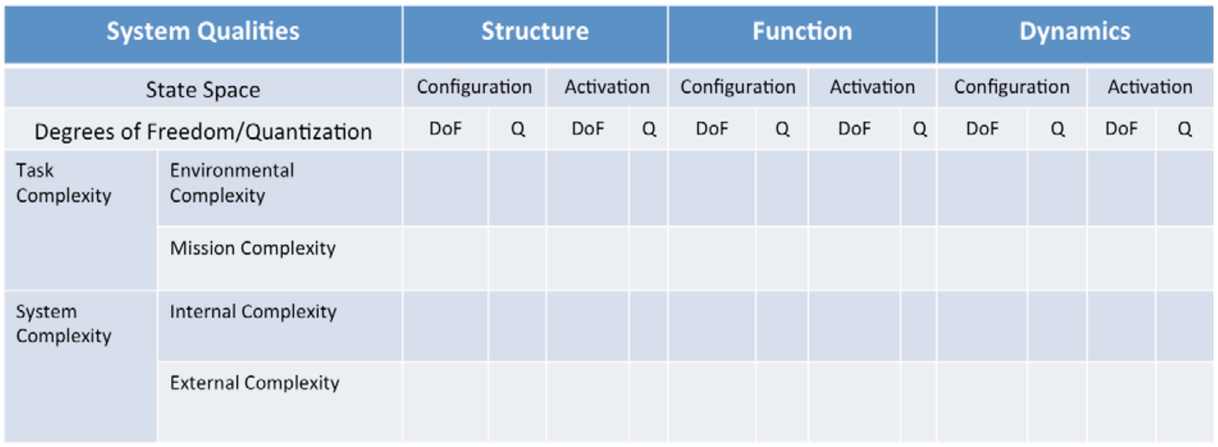
We specifically considered the use of event based processing (e.g., TrueNorth) for context monitoring because of the low power needed for continuous monitoring of sparsely occurring complex events. The low precision, probabilistic, and approximate nature of event based processing techniques is well matched to the nature of contexts in the environment, which do not have precisely definable boundaries or features. The context specific processing component can be implemented using more traditional reconfigurable processing hardware (e.g., GPP, FPGA) that executes synchronously at high rates.

4.0 RESULTS AND DISCUSSION

Below is an enumeration of some of key high level points for program managers and system designers to take into consideration when making decisions regarding the processing capabilities needed for autonomous systems.

- System complexity must be designed to match the requirements of task complexity. Matching task complexity to system resources, or resources to task complexity, is an important part of the design process. The first step in the design process should be a detailed analysis of the task complexity.

Table 7: Components of Complexity Assessments to be Performed During System Design



System Qualities		Structure				Function				Dynamics			
State Space		Configuration		Activation		Configuration		Activation		Configuration		Activation	
Degrees of Freedom/Quantization		DoF	Q	DoF	Q	DoF	Q	DoF	Q	DoF	Q	DoF	Q
Task Complexity	Environmental Complexity												
	Mission Complexity												
System Complexity	Internal Complexity												
	External Complexity												

DoF Stands for Degrees of Freedom; Q stands for Quantization resolution.

- Greater behavioral complexity can provide overall system level energetic advantages that offset the larger processing SWAPC. These impacts must be considered during design when allocating SWAP to processing.
  - For example, the value of pro-active computation is a tradeoff between the cost of pro-active processing and the increase in the value of future actions in achieving the objective (sequential decision making/dynamic programming/model predictive control).
- Behavioral regulation of processing load to match available resources should be considered as an important cognitive function providing a means for autonomous systems to relax static constraints on operating range.
- Task complexity can be addressed by splitting the complexity across the internal and external (human or machine) systems, at the cost of reducing autonomy and introducing dependencies on communications.
  - System designs with dynamically adjustable autonomy should be considered as a compromise that permits resource savings by relying on external systems when communications are available and can move toward progressively greater autonomy as communications degrade.

- The state space formalism provides a common framework for the formulation of both system and task complexity, and can accommodate a wide variety of different computational models including both discrete and continuous systems, and asynchronous and synchronous system.
    - The computation of the energy consumed during end-to-end computations can be partitioned into energy consumed in maintaining state and energy consumed in state transitions.
- Overall complexity consists of distinct contributions from Structural, Functional, and Dynamic complexity. These different contributions to complexity can be traded off against each other.
    - Structural complexity is determined by state space dimensionality, quantization, and topological/geometric complexity.
    - Scaling the task/functional complexity can be addressed in the design of the processing system either by scaling its structural or dynamic complexity (or some combination of the two).
- Specialization of function is an important design strategy for decreasing resource consumption and increasing performance, at the cost of operating range and/or autonomy.
    - It is important when comparing autonomous systems, to compare systems of similar degree of specialization.
- Division of internal system complexity between configuration complexity and activation complexity has important consequences for resource usage. Individual processing architectures are optimized to a particular operating point in this division between configuration/activation complexity.
- In system level end-to-end analysis and implementation, it is convenient to partition task state space into piecewise simple contexts, so that processing requirements for context recognition, context switching, and context specific processing can be accounted for separately.
- 90/10 Rules of Context Sensitive Processing for Autonomy:
    - 90% of the computational resources of autonomous systems are spent in executing the 10% of the code related to context specific processing/activation complexity.
    - 90% of the code in autonomous systems is devoted to monitoring for task relevant context changes that occur 10% of the time (configuration complexity).

5.0 CONCLUSION

Increasing the autonomy of air assets requires the integration of many different cognitive functions currently performed by pilots into real-time end-to-end cognitive processing flows executing on embedded hardware. The growing variety of low SWAP-C specialized processing technologies (e.g., GPU, FPGA, and Neuromorphic) is increasing both the complexity and impact of autonomous system design on system level/mission level efficiency and effectiveness. Design of optimized system level autonomy requires a framework for the performance assessment of entire end-to-end flows, driven by and interacting with operationally realistic environments. In this paper we provide a statespace framework that enables:

- the analysis of a **wide variety of different computational models** including both discrete and continuous systems, and asynchronous and synchronous system.
- the **formulation of complexity of the environment (i.e., physical systems) and complexity of the autonomous system (i.e., computational systems) within the same framework**.
- the description of complexity along three distinct dimensions: **Structure**; **Function**; and **Dynamics**.

This framework is useful from the prescriptive/engineering standpoint in capturing key engineering decision points and requirements and dependencies between engineering decisions from top down perspective. Within this framework it becomes easy to formulate key correspondences (e.g., between task and system complexity, between internal system complexity and external system complexity) and tradeoffs (e.g., between system specialization and autonomy, between structural, function, and dynamic complexity). We apply this framework to the analysis of a proposed context switching cognitive processing architecture which exploits event based processing to efficiently perform context monitoring and context switching, and more conventional processors to perform within context processing. When the task environment consists of many distinct complex contexts, and context switching occurs on a slower timescale (order of magnitude) than update rates needed for within context behaviors, there can be significant advantages to using event based processing hardware in conjunction with conventional processing.

Ultimately, we hope that this analysis framework provides a useful foundation that will facilitate the use of a variety of specialized processing technologies to provide unique advantages in addressing both the strict SWAP constraints on UAV processing and the demanding processing requirements of autonomous behavior in complex and rapidly changing environments.

6.0 REFERENCES

1. Ehsan, Shoaib, and Klaus D. McDonald-Maier, "On-board Vision Processing for Small UAVs: Time to Rethink Strategy," *Adaptive Hardware and Systems, 2009, AHS 2009, NASA/ESA Conference on*, IEEE, 2009.
2. Volpe, Richard, et al., "The CLARAty Architecture for Robotic Autonomy," *Aerospace Conference, 2001, IEEE Proceedings*, Vol. 1. IEEE, 2001.
3. Chong, Hui-Qing, Ah-Hwee Tan, and Gee-Wah Ng, "Integrated Cognitive Architectures: A Survey," *Artificial Intelligence Review*, Vol. 28, No. 2, pp. 103-130, 2007.
4. Cancro, George J. "APL Spacecraft Autonomy: Then, Now, and Tomorrow," *Johns Hopkins APL Technical Digest*, Vol. 29, No. 3, pp. 226, 2010.
5. Huang, Hui-Min, et al., "A Framework for Autonomy Levels for Unmanned Systems (ALFUS)," *Proceedings of AUVSI Unmanned Systems*, 2005.
6. Pratt, Gil, personal communication.
7. Kambadur, Melanie, and Martha A. Kim, "An Experimental Survey of Energy Management Across the Stack," *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications,* ACM, 2014.
8. DeHon, Andre, "Fundamental Underpinnings of Reconfigurable Computing Architectures," *Proceedings of the IEEE*, Vol. 103, No. 3, pp. 355-378, 2015.
9. Blem, Emily, Jaikrishnan Menon, and Karthikeyan Sankaralingam, "Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures," *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013.
10. Murphy, Robin, and James Shields, "The Role of Autonomy in DoD Systems, "*Defense Science Board*, 2012.
11. Vernon, David, Giorgio Metta, and Giulio Sandini, "A Survey of Artificial Cognitive Systems: Implications for the Autonomous Development of Mental Capabilities in Computational Agents," *Evolutionary Computation, IEEE Transactions on*, Vol. 11, No. 2, pp. 151-180, 2007.
12. Soatto, Stefano, "Actionable Information in Vision," *Machine Learning for Computer Vision*, Springer Berlin Heidelberg, pp. 17-48, 2013.
13. Sharon, Yoav, Daniel Liberzon, and Yi Ma, "Adaptive Control Using Quantized Measurements with Application to Vision-only Landing Control," *Decision and Control (CDC), 2010 49th IEEE Conference on*. IEEE, 2010.
14. Wray, R. E., and Christian Lebiere, "Metrics for Cognitive Architecture Evaluation," *Proceedings of the AAAI-07 Workshop on Evaluating Architectures for Intelligence*, 2007.
15. Cassidy, Andrew S., Julius Georgiou, and Andreas G. Andreou, "Design of Silicon Brains in the Nano-CMOS era: Spiking neurons, Learning Synapses and Neural Architecture Optimization," *Neural Networks*, Vol. 45, pp. 4-26, 2013.
16. Tessier, Russell, Kenneth Pocek, and Andre DeHon, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, Vol. 103, No. 3, pp. 332-254, 2015.
17. Trimberger, Stephen M, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," *Proceedings of the IEEE,* Vol. 103, No. 3, pp. 318-331, 2015.

18. Cassidy, Andrew S., and Andreas G. Andreou, "Beyond Amdahl's Law: An Objective Function that Links Multiprocessor Performance Gains to Delay and Energy." *Computers, IEEE Transactions on*, Vol. 6, No. 8, pp. 1110-1126, 2012.
19. Yang, Jisoo, Dave B. Minturn, and Frank Hady, "When Poll is Better than Interrupt," *FAST*, Vol. 12, pp. 3, 2012.
20. Scalera, Stephen M, and Jose R Vazquez, "The Design and Implementation of a Context Switching FPGA," *FPGAs for Custom Computing Machines, Proceedings of the IEEE Symposium on*, pp. 78-85, 1998.
21. Angelini, Chris, and Igor Wallossek, *"Power Consumption - AMD Radeon HD 7970 GHz Edition Review,"* http://www.tomshardware.com/reviews/radeon-hd-7970-ghz-edition-review-benchmark,3232-18.html, June 21, 2012, Accessed 10/ 6/ 2015.
22. Shimpi, Anand Lal, "Power Consumption," January 3 2011, http://www.anandtech.com/show/4083/the-sandy-bridge-review-intel-core-i7-2600k-i5-2500k-core-i3-2100-tested/21, Accessed 10/ 5/ 2015.
23. TechPowerUp, "NVIDIA GeForce GTX 750 Ti 2 GB Review | techPowerUp," February 18 2014,http://www.techpowerup.com/reviews/NVIDIA/GeForce_GTX_750_Ti/23.html, Accessed 10/ 15/ 2015.
24. Altera. n.d., "PowerPlay Early Power Estimators (EPE) and Power Analyzer," https://www.altera.com/support/support-resources/operation-and-testing/power/pow-powerplay.tablet.html, Accessed 10/ 2015.
25. Xilinx. n.d., " Xilinx Power Estimator," http://www.xilinx.com/products/technology/power/xpe.html, Accessed 10/ 2015.
26. Filipp, Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, and Gi-Joon Nam, "TrueNorth: Design and Tool Flow of a 65mW 1 Million Neuron Programmable Neurosynaptic Chip," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pp. 1537-1557, 2015.
27. IBM. n.d., "IBM Research: Neurosynaptic Chips," http://research.ibm.com/cognitive-computing/neurosynaptic-chips.shtml#fbid=YIqItNwpzlt, Accessed 10/ 9/ 2015.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

Acronyms	Description
ALFUS	Autonomy Levels For Unmanned Systems
ARL	Army Research Laboratory
ASIC	Application Specific Integrated Circuit
CISC	Complex Instruction Set Computing
CGRA	Coarse Grained Reconfigurable Architectures
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
CSCPA	Context Switching Cognitive Processing Architecture
DHS	Department of Homeland Security
DoF	Degrees of Freedom
DSP	Digital Signal Processors
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GPU	Graphical Processing Unit
NIST	National Institute of Standards and Technology
RISC	Reduced Instruction Set Computing
RT	Real Time
RTR	Run-Time Reconfiguration
SWAP-C	Size, Weight and Power, and Cost
TDP	Thermal Design Power
UAV	Unmanned Air Vehicle
UMS	Unmanned System
VC	Vapnik-Chervonenkis
VLIW	Very Long Instruction Word